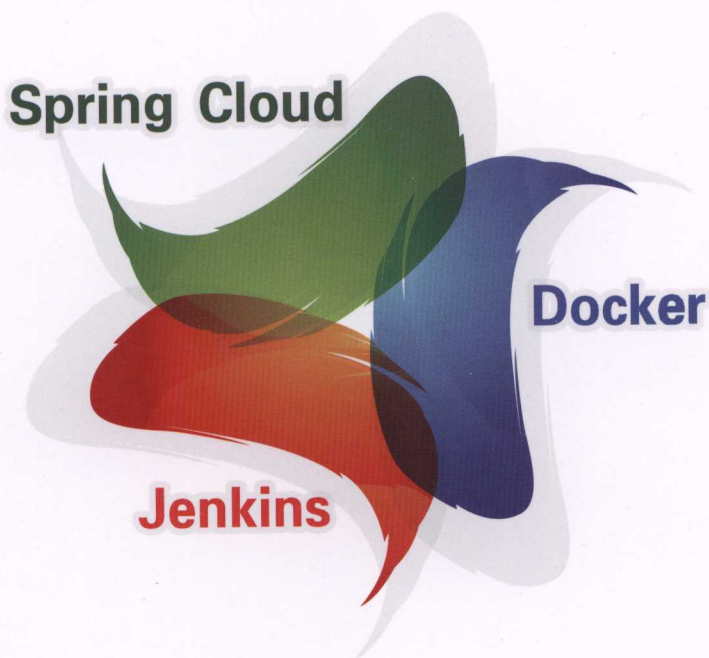


版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Spring Cloud与Docker 高并发微服务架构设计实施

陈韶健 著



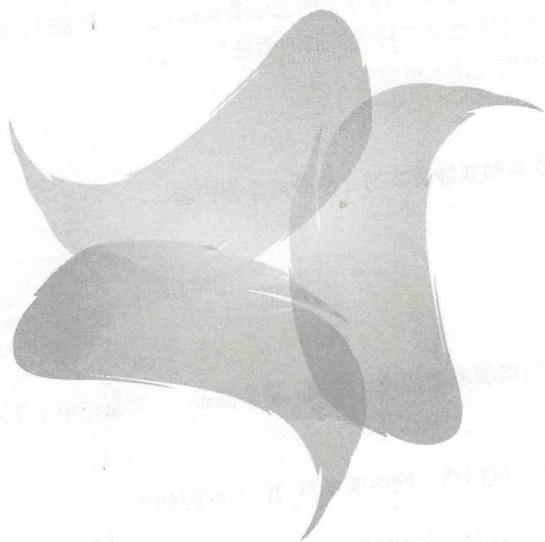
中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

作者简介

陈韶健，资深 IT 技术专家，著有《深入实践 Spring Boot》（2016 年 10 月机械工业出版社）、《Neo4j 全栈开发》（2017 年 6 月电子工业出版社）等书籍，在虚拟化技术领域、数据库使用和大数据分析、分布式架构设计、Spring 等开源框架的使用、微服务实施和开发等领域都有深入的研究和丰富的实践经验。未来研究方向：物联网、智慧城市、AI 人工智能等。



Spring Cloud与Docker 高并发微服务架构设计实施

陈韶健 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

仅供非商业用途或交流学习使用



内 容 简 介

本书从架构设计、应用开发和运维部署三个方面出发,对微服务架构设计的实施进行了全方位的阐述和深入实践,并结合生产实际讲解了 Spring Cloud、Docker 和 Jenkins 等工具的具体使用方法。书中通过一个互联网电商平台实例实现了高并发的微服务架构设计,并通过详细的开发和实施过程,演示了构建一个安全可靠、稳定高效并可持续扩展的系统平台的方法。

本书适合互联网应用开发设计人员参考学习。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Spring Cloud 与 Docker 高并发微服务架构设计实施 / 陈韶健著. —北京: 电子工业出版社, 2018.6

ISBN 978-7-121-34161-8

I. ①S… II. ①陈… III. ①互联网络—网络服务器 IV. ①TP368.5

中国版本图书馆 CIP 数据核字(2018)第 088760 号

策划编辑: 安 娜

责任编辑: 牛 勇

特约编辑: 赵树刚

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 22.5 字数: 504 千字

版 次: 2018 年 6 月第 1 版

印 次: 2018 年 6 月第 1 次印刷

定 价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819, faq@phei.com.cn。



前言

简单地说，微服务就是一种使用轻量级架构设计的组件化的小应用，这种小应用只关注一定范围的业务功能，往往只负责做好一件事情。通过很多这样的小应用，利用一种高效而灵活的管理机制，可以组成一个功能全面且业务丰富的大型平台。这种管理机制包括服务的注册与发现、服务的路由与负载均衡管理、服务之间简单而快捷的通信等功能。

轻量级的微服务架构设计方法，是在竞争激烈的互联网环境中诞生并且发展起来的，非常适合互联网应用快速迭代和规模化扩张的特点。当风靡一时的 SOA 架构因为其重量级的设计方法，并不能适应业务快速变更和高速发展的要求时，微服务架构轻量级的设计风格的出现无疑是软件工程设计的救星，它给软件工程设计重新注入了一股新鲜活跃的血液。

互联网经济激烈竞争的特性，致使业务需求经常发生变化，这就要求技术开发必须具有非常快速的响应能力，以快速抢占市场先机，体现一种“高手过招、唯快不破”的风格。而且，产品一经推出，必须经常进行调整和迭代，以使其始终都能处于“出类拔萃、鹤立鸡群”的地位。更为重要的一面是互联网经济巨大的市场潜力，必然会触发应用产品走上规模化扩张的道路，这种扩张有时甚至是爆炸性的，这对产品的性能和稳定性都提出了前所未有的高要求。

对于软件设计和开发，唯有轻装上阵才能时刻充满活力，始终具有冲锋陷阵的干劲和强壮的生命力。微服务架构轻量级的设计理念及其渐进式的开发特点，正是体现了这种与时俱进的先进设计思想。

在微服务架构设计实施的实践中，Netflix OSS 是大家所公认的最早使用微服务架构设计的典范，Spring 团队在其开源组件的基础上，组建了一个基于 Spring Boot 开发框架的 Spring Cloud 工具套件。Spring Cloud 工具套件为开发者提供了一个完整而配套的工具组件，使微服务架构的实施和开发变得更加普通和容易。Spring Cloud 工具套件提供了包



括但不限于服务注册管理、智能路由、负载均衡服务、熔断容错和服务降级机制、集中配置管理、服务监控与跟踪等功能组件和服务。

本书将详细介绍如何使用 Spring Cloud 工具套件来设计和开发微服务，并且不只是停留在表面上对 Spring Cloud 各个工具组件本身的使用说明和介绍，而是从架构设计出发，说明了如何更好地将微服务架构的设计理念应用于生产实践中，并在实际应用中加深对 Spring Cloud 工具套件的理解和掌握的程度。同时还将介绍怎么使用 Docker 和 Jenkins 等工具来进行微服务的部署和发布，并通过构建一系列高可用性的服务器架构设计，阐述了构建一个稳定、可靠并且能够适应规模化发展的应用平台的方法。

本书将从架构设计、应用开发和运维部署三个方面出发，对微服务架构设计的实施进行全方位的介绍和详细说明，在这一过程中将使用一个互联网平台的实例展开分析和深入实践。

通过对本书的系统学习，可以让读者将微服务架构的设计方法快速应用于生产实践中，为开发团队和企业提供坚不可摧的竞争力。

让我们共同努力，共同探索吧！让先进的技术率先成为我们先进的生产力。

本书的读者对象

本书的读者对象为广大的 Java 开发者、系统架构师和系统运维人员。本书特别适合使用过 Spring 开源框架或具有 Spring 框架基础知识的广大用户群体。

本书章节组成

本书由三部分组成，各部分及其章节的结构如下所示。

第一部分 架构篇

第 1 章 微服务架构与 Spring Cloud

第 2 章 微服务架构最佳设计

第 3 章 电商平台微服务设计实例

第二部分 开发篇

第 4 章 开发工具选用及 Spring Boot 基础

第 5 章 电商平台微服务工程设计

第 6 章 微服务治理基础服务开发

第 7 章 Rest API 微服务开发



第 8 章 Web UI 微服务开发

第 9 章 电商平台移动商城开发

第 10 章 商家管理后台与 SSO 设计

第 11 章 平台管理后台开发

第三部分 运维篇

第 12 章 服务器架构设计与 Docker 使用

第 13 章 数据库集群设计与高可用读写分离实施

第 14 章 分布式文件系统等基础设施安装与配置

第 15 章 使用自动化构建工具 Jenkins 实现 CI/CD

实例代码

本书的实例代码存放在开源中国的码云代码仓库之中,可以通过下列链接打开各个项目工程下载或使用 Git 检出:

<https://gitee.com/chenshaojian/projects>

勘误与反馈

读者如有任何问题,可以通过如下链接发起话题与作者交流。本书在出版印刷之后,有需要勘误的地方也会首先在这里发布:

<https://gitee.com/chenshaojian/SpringCloud/issues>

致谢

非常感谢长期给予我支持和鼓励的朋友们,因为有了你们的支持和鼓励,才让我感到无比的幸福和惬意。还要感谢曾经与我一起进行过微服务开发的小伙伴们,令人欣慰的是,我已经兑现了之前的承诺,将微服务架构设计的经验汇编成书。最后要感谢家人对我的支持,在本书写作的过程中,我几乎将所有的空闲时间都花费在写作之中,而忽略了对你们的陪伴,对此我深感歉意。

如果书中有不对的地方或者任何纰漏,还敬请大家不吝赐教,我将感激不尽。



目 录

第一部分 架构篇

1	微服务架构与 Spring Cloud	2
1.1	微服务概念的由来	2
1.2	微服务的定义	3
1.3	微服务架构与整体式架构的区别	5
1.4	微服务架构与 SOA 的比较	8
1.5	为什么要使用微服务架构	9
1.6	为实施微服务架构做好准备	10
1.6.1	思想观念的转变	10
1.6.2	团队管理方式的改变	11
1.6.3	自动化基础设施的建设	11
1.7	为什么要使用 Spring Cloud	12
1.8	Spring Cloud 组件介绍	13
1.9	Spring Cloud 的版本说明	15
1.10	小结	17
2	微服务架构最佳设计	18
2.1	合理划分微服务	19
2.2	微服务治理	19



2.3	Rest API 微服务设计	21
2.3.1	使用数据库集群.....	22
2.3.2	读写分离设计.....	22
2.3.3	使用缓存.....	22
2.3.4	保证 Rest API 微服务的独立性	23
2.4	Web UI 微服务设计	23
2.4.1	使用 FeignClient 实现负载均衡调用	23
2.4.2	使用 Hystrix 实现容错设计.....	23
2.4.3	使用非阻塞的异步编程技术实现高并发调用.....	24
2.4.4	使用分布式文件系统.....	24
2.5	微服务之间调用规则设计.....	24
2.6	数据最终一致性设计	25
2.7	分布式集群架构设计	26
2.8	微服务运行环境安全设计.....	27
2.9	小结.....	27
3	电商平台微服务设计实例.....	29
3.1	电商平台总体设计.....	29
3.1.1	总体业务流程设计.....	29
3.1.2	总体业务功能设计.....	31
3.2	电商平台业务模型设计	32
3.2.1	移动商城业务模型.....	32
3.2.2	商家管理后台业务模型.....	33
3.2.3	平台管理后台业务模型.....	33
3.3	创建 Rest API 微服务	34
3.4	创建 Web UI 微服务	36
3.4.1	移动商城 Web UI 微服务	36
3.4.2	商家管理后台的 Web UI 微服务	37
3.4.3	平台管理后台 Web UI 微服务	37
3.5	电商平台微服务体系结构.....	38
3.6	小结.....	39



第二部分 开发篇

4 开发工具选用及 Spring Boot 基础41

- 4.1 开发工具选择42
- 4.2 开发环境配置42
- 4.3 创建 Spring Boot 工程43
- 4.4 使用 JPA47
 - 4.4.1 数据源配置48
 - 4.4.2 JPA 配置48
 - 4.4.3 数据实体设计49
 - 4.4.4 存储库接口设计49
 - 4.4.5 单元测试50
- 4.5 使用 Thymeleaf51
 - 4.5.1 控制器设计51
 - 4.5.2 视图设计52
- 4.6 运行与部署52
- 4.7 小结54

5 电商平台微服务工程设计55

- 5.1 微服务工程结构55
- 5.2 电商平台微服务工程组建57
- 5.3 数据库选型57
- 5.4 微服务工程创建步骤58
- 5.5 项目基本配置60
- 5.6 创建模块64
- 5.7 小结66

6 微服务治理基础服务开发67

- 6.1 注册管理中心68
 - 6.1.1 创建注册管理中心68



6.1.2	运行注册管理中心	70
6.1.3	微服务怎样使用注册管理中心	70
6.1.4	构建高可用的注册管理中心	72
6.2	配置管理中心	73
6.2.1	创建配置管理中心	73
6.2.2	微服务如何使用配置管理中心	76
6.2.3	在线更新配置信息	77
6.3	微服务监控中心	78
6.3.1	使用断路器仪表盘实现监控	79
6.3.2	聚合服务监控管理中心	81
6.4	服务跟踪分析中心	86
6.4.1	创建服务跟踪分析中心	86
6.4.2	在微服务中启用服务跟踪功能	90
6.5	日志分析平台	93
6.5.1	创建日志分析平台	93
6.5.2	使用日志分析平台	93
6.6	小结	94

7 Rest API 微服务开发 96

7.1	领域业务开发	96
7.1.1	使用 Druid 数据源	98
7.1.2	JPA 及其配置	100
7.1.3	数据实体建模	101
7.1.4	查询对象设计	104
7.1.5	实体持久化设计	106
7.1.6	持久化测试	107
7.1.7	领域服务开发	109
7.1.8	领域服务的单元测试	111
7.1.9	使用 Redis 实现缓存设计	112
7.2	Rest API 应用开发	117
7.2.1	Rest API 应用配置	117



7.2.2	启动程序设计	119
7.2.3	接口开发	119
7.3	使用消息处理事件	123
7.3.1	消息生产者设计	124
7.3.2	消息消费者设计	125
7.3.3	使用消息测试	128
7.4	小结	129

8 Web UI 微服务开发 131

8.1	高并发接口调用分层设计	131
8.2	通过 FeignClient 调用 Rest API	132
8.3	使用 Hystrix 断路器	134
8.4	使用非阻塞异步编程方法	136
8.4.1	CompletableFuture 介绍	137
8.4.2	性能比较测试	140
8.5	Web 应用开发	145
8.5.1	项目引用配置	145
8.5.2	应用程序配置	146
8.5.3	业务功能开发	148
8.6	开发环境的热部署设置	154
8.7	使用分布式文件系统	157
8.7.1	分布式文件系统客户端开发	157
8.7.2	商品图片上传设计	159
8.7.3	富文本编辑器上传文件设计	160
8.7.4	建立本地文件信息库	163
8.8	小结	166

9 电商平台移动商城开发 167

9.1	移动商城首页设计	168
9.2	使用负载均衡的导航设计	174
9.3	按分类查询设计	176



9.4 商品详情页设计	179
9.5 购买下单实现	181
9.6 用户登录与账户切换设计	184
9.6.1 用户登录设计	184
9.6.2 切换账号设计	186
9.7 订单查询设计	188
9.8 集成测试	191
9.9 小结	192

10 商家管理后台与 SSO 设计 193

10.1 商家权限管理体系设计及开发	194
10.1.1 商家权限体系建模	195
10.1.2 商家权限体系的持久化设计	199
10.1.3 商家权限体系的领域服务开发	201
10.2 商家管理微服务开发	204
10.2.1 商家领域服务层单元测试	204
10.2.2 商家服务的接口开发	208
10.3 SSO 设计	213
10.3.1 SSO 基本配置	213
10.3.2 在 SSO 中使用商家的权限体系	214
10.3.3 用户登录设计	216
10.3.4 有关验证码的说明	218
10.3.5 SSO 的主页设计	220
10.3.6 OAuth2 服务端设计	222
10.4 SSO 客户端设计	224
10.4.1 客户端的项目管理配置	224
10.4.2 客户端的安全管理配置	225
10.4.3 权限验证实现原理	226
10.4.4 如何在应用中接入 SSO	228
10.4.5 有关跨站请求伪造防御的相关设置	230



10.4.6 根据用户权限自动分配菜单	230
10.5 小结	232

11 平台管理后台开发

11.1 平台管理后台领域设计	233
11.1.1 领域实体建模	233
11.1.2 实体的行为设计	236
11.1.3 领域服务开发	236
11.1.4 领域服务单元测试	239
11.2 平台管理后台访问控制设计	240
11.2.1 使用平台管理的用户体系	240
11.2.2 权限管理设计	242
11.3 商家的注册设计	245
11.4 商家菜单体系管理开发	248
11.4.1 分类菜单管理开发	248
11.4.2 模块菜单管理开发	249
11.4.3 访问资源管理开发	252
11.5 商家角色管理开发	255
11.6 小结	257

第三部分 运维篇

12 服务器架构设计与 Docker 使用

12.1 服务器组建	259
12.2 安全的服务器架构设计	260
12.2.1 防火墙安装及配置	260
12.2.2 建立安全的局域网环境	264
12.3 服务器资源分配	266
12.4 CentOS 安装	269
12.4.1 IP 地址设置	270



12.4.2	安全设置.....	270
12.4.3	语言配置.....	270
12.4.4	时间同步配置.....	271
12.5	Docker 和 docker-compose 安装.....	271
12.5.1	Docker 安装及使用.....	272
12.5.2	docker-compose 安装及使用.....	275
12.6	使用 Docker 搭建微服务治理环境.....	279
12.6.1	服务器 1 的部署配置.....	279
12.6.2	服务器 2 的部署配置.....	281
12.7	使用 Docker 部署日志分析平台.....	283
12.8	使用 Docker 部署微服务应用.....	286
12.9	小结.....	286
13	数据库集群设计与高可用读写分离实施.....	288
13.1	MySQL 安装.....	289
13.2	主从同步设置.....	291
13.3	主主同步设置.....	294
13.4	数据库代理中间件选择.....	296
13.5	使用 OneProxy 实现读写分离设计.....	297
13.5.1	OneProxy 安装.....	297
13.5.2	高可用读写分离配置.....	298
13.6	OneProxy 分库分区设计.....	302
13.6.1	按范围分库分表.....	303
13.6.2	按值分库分表.....	303
13.6.3	按哈希算法分库分表.....	304
13.7	双机热备设计.....	306
13.8	小结.....	307
14	分布式文件系统等基础设施安装与配置.....	308
14.1	高可用的分布式文件系统构建.....	308
14.1.1	FastDFS 安装.....	310



14.1.2	跟踪服务器配置.....	310
14.1.3	存储节点配置.....	311
14.1.4	上传文件测试.....	312
14.1.5	Nginx 安装及负载均衡配置.....	313
14.1.6	开机启动设置.....	317
14.2	GitLab 安装.....	322
14.3	Redis 安装.....	324
14.4	RabbitMQ 安装.....	326
14.5	小结.....	327
15	使用自动化构建工具 Jenkins 实现 CI/CD.....	328
15.1	持续交付工作流程.....	330
15.2	Jenkins 安装.....	331
15.3	Jenkins 基本配置.....	333
15.4	Jenkins 自动部署实例.....	335
15.4.1	创建任务.....	336
15.4.2	任务配置.....	337
15.4.3	执行任务.....	340
15.5	小结.....	343
后 记.....		345
参考文献.....		346



第一部分

架构篇

第 1 章 微服务架构与 Spring Cloud

第 2 章 微服务架构最佳设计

第 3 章 电商平台微服务设计实例

这一部分阐述了微服务架构的设计观念及其发展情况，同时介绍了 Spring Cloud 工具套件中各个组件的功能，并说明如何以 Spring Cloud 工具套件为基础，在微服务架构设计中进行权衡与提炼，构建微服务架构的最佳设计，同时通过一个电商平台的设计实例实现了这种最佳设计。



1

微服务架构与 Spring Cloud

近几年，大家都在谈论微服务，微服务是一个非常火爆的关键词，在百度中搜索微服务，随便就有几千万条结果。那么，什么是微服务呢，微服务的概念是怎么产生的呢？

相信大家对微服务也不陌生，或者正在做着相关的开发，现在，我们先来了解一下微服务架构的来龙去脉，为什么要使用微服务架构，微服务架构能给我们带来什么好处，微服务架构与 Spring Cloud 又是一种什么关系。

1.1 微服务概念的由来

据说，早在 2011 年 5 月，在威尼斯附近的一个软件架构师研讨会上，就有人提出了微服务架构设计的概念，用它来描述与会者所看得见的一种通用的架构设计风格。时隔一年之后，在同一个研讨会上，大家决定将这种架构设计风格用微服务架构来表示。

起初，对微服务的概念，也没有一个明确的定义，大家只能从各自的角度说出了对微服务的理解和看法。

有人把微服务理解为一种细粒度的 SOA (Service-Oriented Architecture, 面向服务架构)，一种轻量的组件化的小型 SOA。

有人把微服务看作一种使用 HTTP 通信的自包含的轻量进程。

.....

在这些看法中，比较统一的一点就是，大家都认为微服务是一种小型的应用程序，并



且使用轻量级的设计方法和轻量级的 HTTP 通信。

在 2014 年 3 月，詹姆斯·刘易斯和马丁·福勒所发表的一篇博客中，总结了微服务架构设计的一些共同特点，这应该是一个对微服务比较全面的描述。有关这篇文章的详细内容可从下列链接中打开：

<https://martinfowler.com/articles/microservices.html>。

这篇文章中认为：“简而言之，微服务架构风格是将单个应用程序作为一组小型服务开发的方法，每个服务程序都在自己的进程中运行，并与轻量级机制（通常是 HTTP 资源 API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机器独立部署。这些服务可以用不同的编程语言编写，使用不同的数据存储技术，并尽量不用集中式方式进行管理。”

从这个描述中可以看出，微服务可以是一个小型的服务组件，它使用轻量的 HTTP 协议进行通信。微服务也可以是一个独立的应用程序，它可以具有独立的数据库、独立部署和独立运行。理想的想法是，微服务可以使用不同的语言来编写，并且完全独立自主。

1.2 微服务的定义

从上面对微服务概念形成过程的介绍之中，我们已经对微服务有一个大概的了解，但要说明什么是微服务，很有可能一时也不能说得很清楚。这里，有一点容易混淆的就是微服务架构和微服务，这应该是两个不同的概念，而我们平时一说到微服务，可能已经包含了这两个概念，所以要把它们说清楚难免就会有一种很纠结的感觉。微服务架构是一种设计方法，而微服务应该是指使用这种设计方法而设计的一个应用。所以我们很有必要对微服务的概念做出一个比较明确的定义。

微服务架构是将复杂系统使用组件化的方式进行拆分，并使用轻量通信方式进行整合的一种设计方法。微服务就是通过这种架构设计方法拆分出来的一个独立的组件化小应用。

组件，通常是以代码库的形式，提供函数式调用；而微服务的组件，却以应用的方式，通过使用 HTTP 通信提供接口服务。

微服务架构定义的精髓，可以用一句话来描述，那就是“分而治之，合而用之”。将复杂系统进行拆分的方法，就是“分而治之”的方法。分而治之，可以让复杂的事情变得



简单,这很符合我们平时处理问题的方法。使用轻量通信等方式进行整合的设计,就是“合而用之”的方法。合而用之,可以让微小的力量变得强大。硬件设计中多线程和多任务的技术,可以成倍地提高其并发能力和执行性能。如果软件设计还在使用单线程的方法,那是相当落后的。所以微服务设计中的整合,不但是多服务的整合,也是一种多实例、多副本的整合。通过这种整合,可以充分利用分布式环境的资源和低廉的机器组合成一个强大的服务系统。

微服务轻量级的 HTTP 通信,不同于传统的做法,使用事先设定的 IP 和端口进行访问,而是通过服务注册与发现的机制,使用服务的实例名称进行调用。在这个过程中,服务发现机制将协同路由代理服务 and 负载均衡器一起工作,当客户端使用服务实例名称发出请求时,将通过负载均衡器从服务注册列表中选择一个可用的服务实例,然后才通过实例注册的 IP 和端口路由到相关的服务中。所以提供 API 的微服务,可以发布在任意主机之中,并且可以随时更改主机和端口,也可以发布任意多个副本。

基于上面微服务的定义,我们可以总结出微服务架构设计的几个显著特点,具体如下。

1. 小型化

微服务架构设计的突出之处就是进行服务组件化设计,组件化的结果最显著的特点就是应用程序变小了。使用组件化方式来构建的应用程序,每个组件将只负责完成一定范围的业务功能,更加专一地做好一件事情。因为小型化的特点,会让问题变得更加简单,也使开发变得更加容易。

2. 自治化

使用去中心化的扁平化设计,将使每个服务都能够进行独立自治,这是对复杂功能的一种解耦设计。这种设计的特点也给每个微服务提供了更加自由的管理空间。

每个微服务都是一个独立的应用,独立使用数据库,独立部署,独立运行,这种独立性符合高内聚松耦合的设计原则。在微服务开发和维护中,每个微服务都是独立自治的,一个服务的更新和迭代将不存在对其他任何服务的依赖,同时也不会给其他服务造成影响,或者将这种影响减少到最小限度。

3. 扁平化

独立自治的微服务,可以更加自由地发挥每一个服务的优势和长处。但是这种自由并不是指随意的混搭和组合,而是使用了扁平化的服务治理,让更多的微服务在发挥个性优



势的同时，处在一种杂而不乱的有序可控的状态之中。虽然从整体上微服务已经没有集中管理的概念了，但是微服务在整体上能够发挥更佳的性能优势。

4. 轻量级设计

微服务的组件化特点，也是一种轻量级设计方法的体现，这种轻量级的设计同样体现在微服务的通信设计之中。

微服务的通信设计通常用到两种方式，即使用 API 的同步通信和使用消息通道的异步通信，不管使用哪种通信方式，都没有像 SOA 的 ESB（Enterprise Service Bus，企业服务总线）那样的重量级设计，而是分别使用简单的 REST 协议和轻量的消息总线来实现。

5. 渐进式设计

一个产品从成型到成熟总是要经历一个过程，这个过程就是渐进式设计的特点。由于微服务小型而独立的特点，微服务设计可以使用业务驱动的方式进行，使用快速迭代进行不断地修正和调整，以使产品趋于成熟。

1.3 微服务架构与整体式架构的区别

如果是一个小型项目，使用整体式架构来设计，其好处是明显的，因为它的设计、开发、测试和部署，都在一个项目上就可以完成。

如果一个业务复杂的大型项目，也使用整体式架构来设计，就将存在很多问题。可能刚开始的阶段，还感觉不到什么，但是随着时间的推移，加入的功能越来越多，一个项目就变成了一块巨大的石头，十分笨重。

面对一个如此巨大的项目，开发人员要弄清楚它的代码逻辑，必须要花费很多的时间。而针对某一项功能的更改，极有可能动一线而牵全身，这会让实施的人员变得很难应对。所以这种项目将会变得越来越难以维护，越来越不便更新。

整体式架构的稳定性也不能得到有效的保障，如果其中的一个模块出现问题，将会影响到整个系统的正常运行，甚至造成整个系统的崩溃。而要进行问题的跟踪，因为系统庞大，往往难上加难。

另外，一个巨大的应用项目，也不方便进行持续开发，它不能适应需求的变更，不能适应快速迭代的敏捷开发方法，所以这样的项目最终就变成了业务发展的绊脚石。



相比之下，大型项目使用微服务架构就具有明显的优势。

微服务架构设计，就是将复杂事情进行简单化处理的方法，它将一个复杂的系统，拆分成一些小型的应用来开发，起到了一种“大事化小，小事化无”的效果。因为简单，代码的逻辑会变得更加清晰，这无疑减轻了程序员繁重的劳动；因为简单，所以能够专注，能够将每一件事情做好，做到极致。

微服务中独立的小型应用，也非常适合使用敏捷开发方法，能够快速响应需求的变化，进行快速更新，快速迭代，甚至将一个应用推倒重来也是很容易做到的。

因为每个微服务都是独立自治的，一个服务的故障也不会影响到全局系统的正常运行，或者说可以将这种影响降到最低限度。况且，微服务架构的容错设计可以避免这种情况发生。

微服务架构高可用的特点是系统稳定性的最好保障，而且微服务能够支持高并发的调用，支持高流量的访问，能够持续保证平台规模化发展的要求，这是整体式架构所不能做到的。

如果我们使用一个六边形结构来表示整体式架构的话，将可以绘制出如图 1-1 所示的结构图。

这个六边形的核心是整体式架构的领域业务模型，它通过系统接口使用各种适配器，例如数据库适配器、文件适配器等，与外部管理系统进行连接。然后通过接口使用诸如 Rest API 适配器、Web UI 适配器等对外部 App 和终端用户提供接口调用和 Web 访问等服务。

从图 1-1 中可以看出，整体式架构是一个大而全的系统。在微服务架构设计中，我们可以使用一个小六角形来表示每个微服务，它相当于将整体式架构进行拆分之后得到的结果，如图 1-2 所示。

小六角形的微服务同样使用接口，通过各种适配器来连接外部管理系统，而微服务之间也可以通过接口，使用 Rest API 适配器进行通信，而对于 App 和终端用户，将分别由不同的微服务提供相应的适配器及其服务。



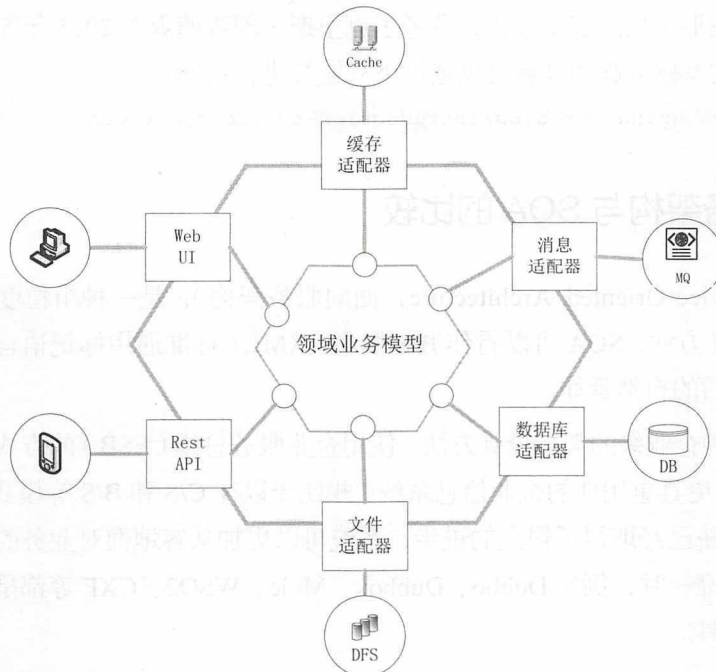


图 1-1 整体式架构六边形结构图

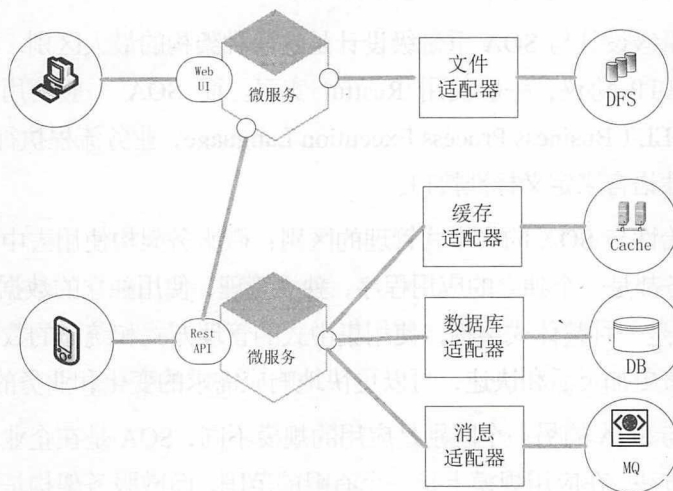


图 1-2 微服务架构结构图

从上面这两种结构图形的比较中，可以非常明显地看出整体式架构与微服务架构的巨大区别。

有关这种图形结构的表示方式，参考了克里斯·理查德森在 2015 年 5 月发表的系列文章，对这些文章感兴趣的读者可以通过下列链接进行访问：

<https://www.nginx.com/blog/introduction-to-microservices/>

1.4 微服务架构与 SOA 的比较

SOA (Service-Oriented Architecture, 面向服务架构)，是一种粗粒度、松耦合的面向服务架构设计方法。SOA 可以看作 B/S 模型、XML (标准通用标记语言的子集)/Web Service 技术之后的自然延伸。

SOA 是一种企业级的架构设计方法，使用企业服务总线 (ESB) 的方式来构建一个更高效、更可靠、更具重用性的企业信息系统。相比于以往 C/S 和 B/S 等模式的设计，使用 SOA 架构的系统已经取得了很大的进步，系统可以更加从容地面对业务的急剧变化。所以 SOA 曾经风靡一时，例如 Dubbo、Dubbox、Mule、WSO2、CXF 等都是有一些比较优秀的 SOA 开源工具。

微服务架构与 SOA 在表面看来还是有一点相似的，以至于早期有人会认为微服务是一个细粒度的 SOA，其实它们的区别还是很大的。

微服务的轻量级设计与 SOA 重量级设计是这两种架构的最大区别。微服务的通信设计使用简单的 HTTP 协议，一般使用 Restful 实现。而 SOA 一般使用复杂的协议，如 WebService 或 BPEL (Business Process Execution Language, 业务流程执行语言) 等，还需要使用服务描述性语言来定义标准接口。

微服务的自治性与 SOA 的集中式管理的区别：微服务架构使用去中心化的扁平化管理方式，每个服务都是一个独立的应用程序，独立管理，使用独立的数据库，独立部署和独立运行。SOA 是一种整体式架构，使用集中式的管理方式和统一的数据中心。所以微服务的开发和部署更加灵活和快速，可以更快地响应需求的变化和业务的更新。

微服务架构与 SOA 的另一个区别是应用的规模不同，SOA 是在企业计算领域中产生的一种架构设计方法，在应用规模上是一个有限的范围，而微服务架构是产生于互联网环境中的一种设计方法，它的设计更能适应无限广阔的环境，更能适应互联网高流量、高并发的规模扩张。

微服务架构的高可用设计、自由伸缩性、负载均衡、故障转移等特性是 SOA 设计不



够重视的地方。微服务的高可用设计通过微服务治理，为每个微服务的管理和部署提供一个可以扩展的无限广阔的空间，它可以表现为一个三维结构，如图 1-3 所示。

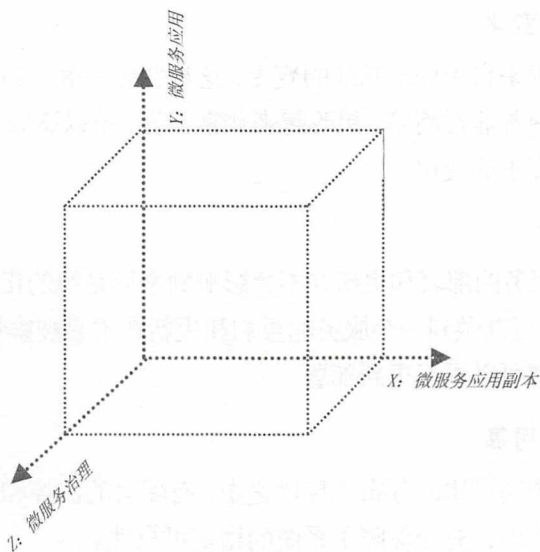


图 1-3 微服务治理的三维结构

在这个三维结构中，如果我们用 Y 轴表示微服务应用，用 X 轴表示微服务应用部署的多个副本，那么用 Z 轴表示微服务治理，它将提供服务路由和负载均衡管理等功能，并且还可以提供分区管理的功能。

1.5 为什么要使用微服务架构

从上面的比较可以看出，整体式架构已经不再适合于一个大型项目或者一个应用平台的开发，而 SOA 架构虽然曾经风靡一时，但是其重量级的设计也已经成为快速开发的障碍，所以这两种架构设计方法都将被微服务架构所取代。微服务架构轻量级的设计风格，不管从理论上还是从技术实现上，已经越来越多地得到更多人们的肯定和认可，大家对它的未来发展趋势都抱有一种乐观的态度。使用微服务的好处如下。

1. 开发简单

微服务架构将复杂系统进行拆分之后，让每个微服务应用开发都变得非常简单，没有

太多的累赘。对于每一个开发者来说，这无疑是一种解脱，因为再也不用进行繁重的劳动了，每天都在一种轻松愉快的氛围中工作，其效率将会成倍地提高。

2. 快速响应需求变化

一般的需求变化都来自于局部功能的变更，这种变更将落实到每个微服务上，而每个微服务的功能相对来说都非常简单，更改起来非常容易，所以微服务非常适合敏捷开发方法，能快速响应业务需求的变化。

3. 随时随地更新

一方面，一个微服务的部署和更新并不会影响到全局系统的正常运行；另一方面，使用多实例的部署方法，可以做到一个服务的重启和更新在不易被察觉的情况下进行。所以，每个微服务任何时候都可以进行更新部署。

4. 系统更加稳定可靠

微服务运行在一个高可用的分布式环境之中，有配套的监控和调度管理机制，并且还可以提供自由伸缩的管理，充分保障了系统的稳定可靠性。

5. 规模可持续扩展

每个互联网应用都具有巨大的市场潜力，一旦这种潜力被激发，就需要系统能支持大规模的高并发访问机制，使用微服务架构设计的系统，将能适应业务的快速增长，并可持续支持规模化的扩展。

1.6 为实施微服务架构做好准备

相比于以往的设计方法，微服务是一个全新的设计理念，为了能够顺利地实施微服务架构设计，必须在前期做好充分的准备，在思想观念、团队管理和运维管理上进行相应的变革。

1.6.1 思想观念的转变

开始进入微服务领域，必须从做项目的观念中转变到做产品的观念中来。如果是一个软件项目，在完成了业务需求的设计之后，最终交付使用，其项目开发的生命周期即宣告结束。而做产品就完全不一样，只要产品成型上线，产品有存在的价值，开发将永远没有



终结，随着产品的更新换代，其中的应用程序和组件就要跟着不断地更新和迭代。

微服务架构中的渐进式设计的特点，就是一种做产品的观念的真实体现。一方面，一个产品的最初成型设计，由于种种原因并不可能把所有功能都考虑得很周到，这需要一定时间进行慢慢磨合；另一方面，市场总是在不断变化，所以产品的业务功能也应该随着时间的推移而发生一定的变化。

做产品的观念将贯穿于一个系统平台的整个生命周期之中，并随着平台的发展和演化，最终将产品打成一个生态体系。

1.6.2 团队管理方式的改变

传统的团队管理是按技术来进行分组的，在一个开发团队中，可能有 UI 设计组、前端开发组、后端开发组、测试组和运维组等的划分。

微服务架构的实施将按业务功能进行划分，所以对团队的管理最好应该以业务来进行分组，将产品设计、前端开发、后端开发、测试和运维等人员围绕业务功能分配在同一组中，这样不但可以增强团队的凝聚力量，而且也可避免将大量的时间浪费在不同组别的沟通和工作协调之上。

在实际的操作中，因为前端开发和运维管理的消耗并不是很大，所以这两部分的人员，可以进行机动调整，但这种调整最好也是在业务相近的领域中进行，并保持一定的连贯性，即原来由谁负责的工作，在更新和维护中还是由他来负责。

为了减少资源的浪费和增加每个人员的工作饱和度，一个业务小组往往并不只负责一个微服务，有可能会负责两三个微服务的开发，这主要看微服务划分的粗细粒度来定。

1.6.3 自动化基础设施的建设

从整体式架构向微服务架构转变之后，项目会变得多起来，并且迭代的周期会越来越短，所以对测试和运维将在很大程度上提出更高的要求，并且其工作量也会比以前变得更多，所以单纯依靠人力来完成这两部分的工作是远远不够的，这就要求必须有自动化基础设施的支持，来完成自动集成、自动测试，以及持续交付、持续部署的工作。

一个原来可能几个项目就能支撑起来的应用平台，使用微服务架构进行拆分之后，可能会变成几十个项目，甚至上百个项目。所以如果还像原来那样分配测试和运维工作，将



需要增加更多的人员配备。

在服务器资源的使用上，也相应地有所增加，但是因为每个微服务应用所占用的资源并不是很大，可以在原来的服务器中使用虚拟机技术来扩展服务器群组。对于微服务的部署，我们将主要使用 **Docker** 容器引擎来实施，这可以非常自由地将微服务分散部署在分布式环境之中。这样也就能更好、更有效地利用服务器的资源。

1.7 为什么要使用 Spring Cloud

微服务架构的实施和使用已经经历了一定的历程，在这一过程中，亚马逊、Netflix 在使用微服务架构设计中成功的经验及其取得的成绩，是一个令人振奋的结果。特别是 Netflix OSS 开源组件的推出，更掀起了一番微服务的热潮。

Spring 团队在 Netflix OSS 的基础上，推出了 Spring Cloud 微服务开发工具套件，极大地降低了微服务开发的门槛。使用 Spring Cloud 工具套件，可以让一个 Java 开发者，非常容易地开发微服务应用。

Spring Cloud 专注于为典型用例和可扩展机制提供良好的开箱即用体验，涵盖了如下各个方面的功能。

- 分布式/版本化配置。
- 服务注册和发现。
- 路由。
- 服务之间调用。
- 负载均衡。
- 断路器。
- 全局锁。
- 主机选举和集群状态。
- 分布式消息。

有了 Spring Cloud，开发微服务应用就变得非常简单了。使用 Spring Cloud，你可以在任何环境中进行开发和调试，包括自己的笔记本电脑、公司的局域网环境以及 Cloud Foundry 等托管平台。



1.8 Spring Cloud 组件介绍

1. Spring Cloud Config

配置管理工具包，可以将每个服务的配置放到远程服务器，实现集群化的集中管理，目前支持本地存储、Git 以及 Subversion 等。

2. Spring Cloud Netflix

包含 Netflix OSS 的一些基础组件，如 Eureka、Hystrix、Zuul、Archaius、Ribbon、Turbine 等。

- Eureka 是云端服务发现，用于云端服务注册与定位，以实现云端服务发现和故障转移等服务治理。
- Hystrix 是一个集断路器、容错机制、降级机制等功能于一体的管理工具，通过这一工具可以实现对第三方库的延迟和故障转移提供全面监测和调控能力。
- Zuul 是在云平台上提供动态路由、监控、安全等边缘服务的管理框架。Zuul 相当于设备和微服务应用的 Web 网站后端所有请求的前门。
- Archaius 是一个配置管理 API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。
- Ribbon 提供云端负载均衡管理，有多种负载均衡策略可供选择，能自动配合服务发现和断路器使用。
- Turbine 是聚合服务发送事件流数据的一个工具，使用可配置方式用来监控集群中服务的运行情况。

3. Spring Cloud Bus

一个事件、消息总线，用于在集群中传播状态变化，可与 Spring Cloud Config 联合使用，实现动态配置管理。

4. Spring Cloud for CloudFoundry

通过配置协议绑定服务到 CloudFoundry，CloudFoundry 是 VMware 推出的开源 PaaS 云平台。

5. Spring Cloud CloudFoundry Service Broker

提供一个起点，为 CloudFoundry 建立托管服务代理。



6. Spring Cloud Cluster

提供 Leadership 选举，如 Zookeeper、Redis、Hazelcast、Consul 常见状态模式的抽象和实现。

7. Spring Cloud Consul

封装了 Consul 操作，Consul 是一个服务发现与配置工具，与 Docker 容器可以无缝集成。

8. Spring Cloud Security

基于 Spring Security 的安全工具包，可以为应用添加安全控制。

9. Spring Cloud Sleuth

日志收集工具包，封装了 Dapper 和 log-based 追踪，以及 zipkin 和 HTrace 操作，为微服务应用实现了一种分布式追踪解决方案。

10. Spring Cloud Data Flow

大数据操作工具，作为 Spring XD 的替代产品，它是一个混合计算模型，结合了流数据与批量数据的处理方式。

11. Spring Cloud Stream

数据流操作开发包，封装了与 Redis、Rabbit、Kafka 等发送和接收消息的方法。

12. Spring Cloud Stream App Starters

提供数据流基于 Spring 的与外部系统集成方法。

13. Spring Cloud Task

提供云端计划任务管理与调度。

14. Spring Cloud Zookeeper

连接 Zookeeper 的工具包，用于使用 Zookeeper 方式的服务注册、发现与管理。

15. Spring Cloud Connectors

便于云端应用程序在各种 PaaS 平台连接到后端，如数据库和代理服务等。



16. Spring Cloud Starters

使用 Spring Boot 方式的启动项目工具包, 为 Spring Cloud 提供开箱即用的依赖管理。

17. Spring Cloud CLI

基于 Spring Boot CLI, 可以使用命令方式快速建立云应用。

18. Spring Cloud Contract

这是一个使用模拟测试的总体项目, 其中包含帮助用户成功实施消费者驱动合同方法的解决方案。目前由 Spring Cloud Contract Verifier 项目组成。

1.9 Spring Cloud 的版本说明

截至本书定稿时, Spring Cloud 及其组件版本对照如表 1-1 所示。

表 1-1 Spring Cloud 及其组件版本对照表

Component	Camden.SR7	Dalston.SR4	Edgware. RELEASE	Finchley. M4	Finchley.BUILD -SNAPSHOT
spring-cloud-aws	1.1.4.RELEASE	1.2.1.RELEASE	1.2.2.RELEASE	2.0.0.M2	2.0.0.BUILD-SNAPSHOT
spring-cloud-bus	1.2.2.RELEASE	1.3.1.RELEASE	1.3.2.RELEASE	2.0.0.M3	2.0.0.BUILD-SNAPSHOT
spring-cloud-cli	1.2.4.RELEASE	1.3.4.RELEASE	1.4.0.RELEASE	2.0.0.M1	2.0.0.BUILD-SNAPSHOT
spring-cloud- commons	1.1.9.RELEASE	1.2.4.RELEASE	1.3.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-contract	1.0.5.RELEASE	1.1.4.RELEASE	1.2.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-config	1.2.3.RELEASE	1.3.3.RELEASE	1.4.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-netflix	1.2.7.RELEASE	1.3.5.RELEASE	1.4.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-security	1.1.4.RELEASE	1.2.1.RELEASE	1.2.1.RELEASE	2.0.0.M1	2.0.0.BUILD-SNAPSHOT
spring-cloud- cloudfoundry	1.0.1.RELEASE	1.1.0.RELEASE	1.1.0.RELEASE	2.0.0.M1	2.0.0.BUILD-SNAPSHOT
spring-cloud-consul	1.1.4.RELEASE	1.2.1.RELEASE	1.3.0.RELEASE	2.0.0.M3	2.0.0.BUILD-SNAPSHOT
spring-cloud-sleuth	1.1.3.RELEASE	1.2.5.RELEASE	1.3.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-stream	Brooklyn.SR3	Chelsea.SR2	Ditmars.RELEASE	Elmhurst. M3	Elmhurst.BUILD-SNAPS HOT
spring-cloud- zookeeper	1.0.4.RELEASE	1.1.2.RELEASE	1.2.0.RELEASE	2.0.0.M3	2.0.0.BUILD-SNAPSHOT



续表

Component	Camden.SR7	Dalston.SR4	Edgware.RELEASE	Finchley.M4	Finchley.BUILD-SNAPSHOT
spring-boot	1.4.5.RELEASE	1.5.4.RELEASE	1.5.8.RELEASE	2.0.0.M6	2.0.0.BUILD-SNAPSHOT
spring-cloud-task	1.0.3.RELEASE	1.1.2.RELEASE	1.2.2.RELEASE	2.0.0.M2	2.0.0.RELEASE
spring-cloud-vault		1.0.2.RELEASE	1.1.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT
spring-cloud-gateway			1.0.0.RELEASE	2.0.0.M4	2.0.0.BUILD-SNAPSHOT

其中，Spring Cloud 的版本号为了与各个组件的版本号区分开来，使用了伦敦地铁站的名字来命名，并按字母顺序排列，截至目前，累计的版本有：Angel、Brixton、Camden、Dalston、Edgware、Finchley 等，本书实例将使用上表中第三个发行版：Edgware.RELEASE，这是一个稳定版，对应的 Spring Boot 版本为 1.5.9 发行版。上表中的“M1”、“M2”等是指各个里程碑（Milestone）版本。

Spring Cloud 的版本更新可留意其官方发布的信息：

<http://projects.spring.io/spring-cloud/>

打开上面链接，我们可以查看 Edgware 版本的 POM（Project Object Model）配置，如图 1-4 所示。

The screenshot displays the Spring Cloud website for the Edgware version. It includes a 'Quick Start' section with instructions on using the release train label. Below this is a 'Download' section with a dropdown menu set to 'Edgware' and buttons for 'MAVEN' and 'GRADLE'. The main content area shows the recommended POM configuration for getting started with Spring Cloud in a project. To the right, there is a 'Sample Projects' list and a 'References' section with links to other versions like Finchley M4, Dalston SR4, and Camden SR7.

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId></groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>
  <dependency>
    <groupId></groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
</dependencies>

```

图 1-4 Spring Cloud 版本信息

当我们在一个工程中使用像图 1-4 一样的 POM 配置之后（如下所示），在工程中引用 Spring Cloud 的其他组件时就可以不用再指定版本号，它将根据表 1-1 的对应关系适配一个组件合适的版本。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

1.10 小结

本章介绍了微服务架构的基本概念及其特点，并且通过与整体式架构和 SOA 架构的比较，加深了对微服务的理解，同时也简要介绍了 Spring Cloud 工具套件中各个组件的功能，从中可以看出，使用微服务架构将使我们面对复杂系统时能够将工程设计变得更加简单，并且能够快速而从容地响应频繁的需求变更，而 Spring Cloud 配套和强大的功能，将使微服务的开发更加容易和高效。

下一章将介绍如何使用 Spring Cloud 工具来进行微服务架构设计，开始进行软件工程设计中的一种全新体验。



2

微服务架构最佳设计

从上章微服务架构的介绍中，我们知道，使用微服务架构有很多好处，并且在各个方面都有其独特的优势，但是，每一件事情都不是绝对的，使用微服务架构同时也充满着挑战。因此，我们必须在开始进行微服务架构设计时，进行全盘考虑，权衡利弊，才能做出合理的抉择，取得最佳的设计效果。

在微服务架构设计中，对复杂系统进行拆分之后，会不会产生一些新的问题呢？比如微服务之间的相互调用和通信会不会很复杂？由于每个微服务都有独立的数据库，那么分散的数据管理怎么保证数据的一致性？如果单个微服务的功能变更，会不会影响到多个微服务的正常运行？诚然，这些问题是确实存在的。在克里斯·理查德森所发表的有关微服务架构设计的博客文章中，在这方面也有详细的描述，总结起来包括如下几点：

- 微服务的粗细粒度不好把握。
- 分布式的微服务增加了服务之间相互调用及其通信的复杂性。
- 分散的数据管理难以保证数据的一致性。
- 由多个微服务组成的系统会增加集成测试的复杂性。
- 单个服务的变更可能会影响到多个服务。
- 部署的复杂性。

下面将说明如何通过合理的设计，不但可以解决上面提到的问题，并且能充分发挥微服务架构的优势，以达到一种扬长避短的目的。



2.1 合理划分微服务

微服务架构设计的首要任务就是怎么合理地划分微服务,即怎么围绕业务功能来创建微服务。在进行微服务划分时,有关微服务粗细粒度的考虑,建议在平台创建的初始阶段,可以使用粗粒度的方法按业务功能进行划分。然后,随着业务的发展及其运营的情况,再依据发展规模考虑是否继续细分。下面将使用水平划分法和垂直划分法两种方法相结合的方式来创建微服务。

一方面,在水平方向上,按业务功能不同来划分微服务,并把这次划分所创建的微服务称为 Rest API 微服务。Rest API 微服务负责业务功能的行为设计,主要完成数据管理方面的工作,并通过使用 Rest 协议对外提供接口服务。

另一方面,在垂直方向上,再以 Rest API 微服务为基础实现前后端分离设计,创建 Web UI 微服务。Web UI 微服务不直接访问数据,而只专注于人机交互界面的设计,它的数据存取将通过调用 Rest API 微服务来完成。

这样,经过两次微服务划分,我们就可以创建出 Rest API 和 Web UI 两种类型的微服务,也就是说,我们只要使用两种类型的微服务,就可以构成一个复杂的业务系统。

使用 Rest API 和 Web UI 微服务,再结合高性能和高并发的设计,并通过微服务的多副本发布,就可以构成一个能适应任何规模访问的多维的稳定牢固的网格结构,并且这个网格结构还具有自由伸缩的特性,可以根据业务的发展规模进行缩编或者进行扩充,这样也就可以非常容易地搭建一个可持续扩展的系统平台。

2.2 微服务治理

微服务架构的实施,将会产生出越来越多的微服务,而微服务的独立自治性及其设计的扁平化管理方法,都给每个微服务的开发、部署和更新提供了更大的灵活性,所以微服务的运行环境、服务间的相互调用将会变得越来越复杂。因此必须有一些组件和服务来对运行中的微服务进行有效治理,才能在一个庞大的分布式环境中保证每个微服务的运行和服务间的调用处在一种有序而不杂乱、稳定而高效的状态之中。

Spring Cloud 工具套件使用了基于 Netflix OSS 的一些基础组件来实现微服务治理,这些组件主要包括:



- 注册管理服务组件 Eureka，提供服务注册和发现的功能。
- 负载均衡服务组件 Ribbon，提供负载均衡调度管理的功能。
- 边缘代理服务组件 Zuul，提供网关服务和动态路由的功能。
- 断路器组件 Hystrix，提供容错机制、服务降级、故障转移等功能。
- 聚合服务事件流组件 Turbine，可用来监控集群中服务的运行情况。
- 日志收集组件 Sleuth，可以通过日志收集提供对服务间调用进行跟踪管理的功能。
- 配置管理服务组件 Config，提供统一的配置管理服务功能。

这些组件是如何进行微服务治理的呢？如图 2-1 所示为一个微服务治理主要过程的序列图，可以用来简要说明微服务治理的工作原理。在这个序列图中，Eureka 管理每个注册的微服务实例，并为其建立元数据列表，当一个服务消费者需要调用微服务时，Ribbon 将依据微服务的实例列表实行负载均衡调度，这种调度默认使用了轮询算法，它将从实例列表中取出一个可用的实例，然后 Zuul 依据实例的元数据，对服务进行路由，在路由的过程中，Hystrix 将检查这个微服务实例的断路器状态，如果断路器处于闭合状态，即提供正常的服务；当断路器打开时，说明服务已经出现故障，Hystrix 将根据实例的配置情况实行故障转移、服务降级等使用机制。

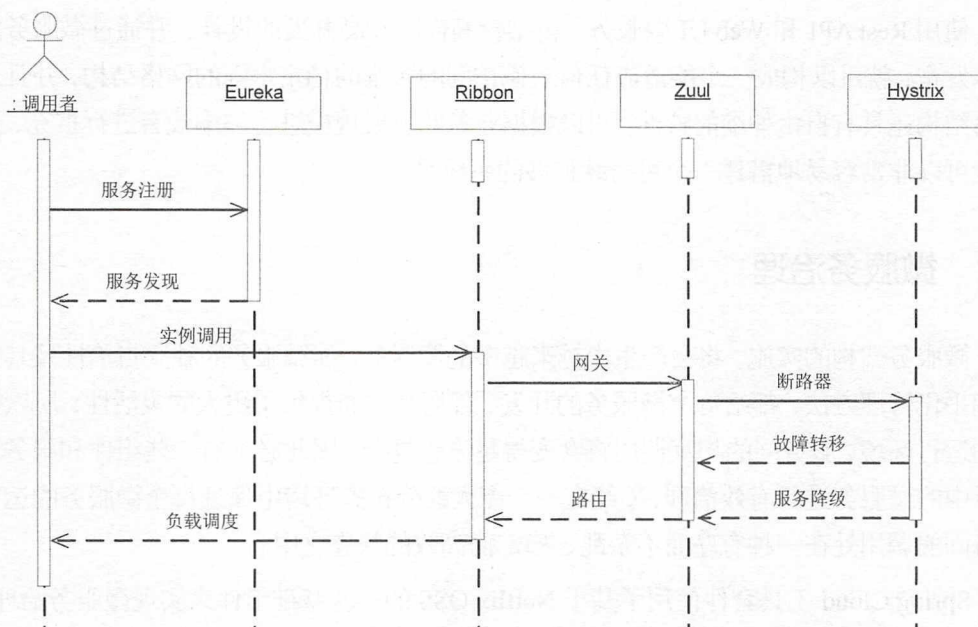


图 2-1 Spring Cloud 微服务治理序列图

另外，使用 Turbine 组件可以对运行中的微服务实现全面监控，使用 Config 可以构建一个在线更新的配置管理中心，使用 Sleuth 的日志收集功能，结合 Zipkin 的使用可以组建一个跟踪服务器。通过这些组件和服务的使用，将可进一步加强微服务治理的力度。

所以，通过 Spring Cloud 的工具组件进行微服务治理，微服务的运行就将处在一种有序可控的范围之中，微服务之间的相互调用和通信就会变得非常简单，并且快速而高效。

2.3 Rest API 微服务设计

通过围绕业务功能使用水平划分方法划分出来的 Rest API 微服务是一个独立的小应用，这种小应用具有独立的数据库，可以独立部署和独立运行。也就是说，每个 Rest API 微服务是完全独立自治的。

Rest API 微服务需要实现两方面的功能：一方面，对内实现数据管理的功能；另一方面，对外提供 API 调用。

有关数据管理的功能，我们将使用 Spring-Data-JPA 来开发。JPA(Java Persistence API)使用存储库(Repositories)的方式来实现数据的持久化，可以简化很多开发工作。另外 JPA 面向对象的特点也适合使用领域驱动设计的方法进行建模，这些都符合微服务架构轻量级设计的特点。

接口的开发将根据 Restful 规范，使用如下所示的方法对外提供相应的数据服务：

- GET
- POST
- PUT
- DELETE

这些接口方法已经涵盖了数据的新增、查询、更新和删除等操作。

Rest API 微服务的实现所体现出来的设计容易、开发简单、访问高效的特点，非常适合使用敏捷开发方法，也可以适应快速迭代和多副本发布的要求。

在我们的微服务架构设计中，Rest API 微服务就相当于数据库调用的一个中间件，它的调用性能将对整个系统的性能起到决定性的作用。所以，高性能的 Rest API 微服务设计，就是微服务架构最佳设计中的一个重点所在。高性能的 Rest API 微服务设计，将主要通过下述一些方法来实现。



2.3.1 使用数据库集群

在数据库集群设计上,我们将使用一种可扩展的分布式架构来实现,这种架构使用集群和分组设计,可以实现根据规模的发展情况提供可持续扩容的特性。

数据库的集群设计,主要通过主从同步的方式来扩展数据库的访问性能,通过建立多个只读的从机来缓解主机的读写压力。

集群的分组设计是分布式环境中一种高可用的设计,在一般情况下,可能只有一个分组提供正常的服务,其他分组只是充当冗余备份的角色,当提供正常服务的分组负载过高时,处于备份状态的分组才变成提供正常服务的角色。

这样,通过集群和分组相结合的方式,就可以充分提高数据库的访问性能。

2.3.2 读写分离设计

使用数据库集群就可以进行读写分离设计。通过读写分离设计可以充分调用数据库集群和分组的资源。读写分离的设计将由数据库代理中间件来实现。数据库代理不但实现了读写分离的功能,也充当了集群分组负载均衡调度的角色。另外,数据库代理也为应用访问集群数据库,提供了一种完全透明的使用方法,即对于一个应用来说,它还是像以往使用单机数据库一样使用连接池,它并不清楚这种连接池里面还有一个巨大的集群体系。

另外,在读写分离设计中,还可以将数据存取进行分区设计,即使用分库分区表的方法,可以对单一表格的存储容量进行扩充,同时通过分区表设计,也能提高大容量数据的读取效率。

2.3.3 使用缓存

使用数据缓存是一种提高数据的存取性能的方法,同理也是我们解决数据库主从设计中可能出现同步延迟的有效方法。

我们将使用 Redis 来实现缓存设计,Redis 是一个高性能 NoSQL 数据库,并且具有非常高效的存取性能。Redis 首先使用内存作为数据存取的缓冲区,同时也能将数据做持久化保存在磁盘中。

为了提高缓存的命中率,在缓存设计中必须遵循以下两点规定:



第一，不要在缓存中存取大容量的数据。

第二，合理设置每一个缓存数据的有效期，不将缓存数据做持久化保存。

2.3.4 保证 Rest API 微服务的独立性

Rest API 微服务是一个独立的应用，并且具有独立自治特性，但是如果在 Rest API 微服务之间进行互相调用，就将破坏这种独立性。所以，禁止在 Rest API 微服务之间进行互相调用，可以充分保证 Rest API 微服务的独立性，让 Rest API 微服务发挥最佳的性能效应。

2.4 Web UI 微服务设计

Web UI 微服务是一个前端设计，它也可以使用 Node.js、Angular.js 或 Vue.js 等前端设计工具来进行开发。这里推荐使用 Spring-Boot-Web 来开发 Web UI 微服务，因为这样会更加简便和快速，并且与 Spring Cloud 工具也能够取得更多的默契和更多方面的搭配。

Web UI 微服务的设计，包含了两个方面的功能：一方面实现对 Rest API 微服务的调用；另一方面是用户界面设计。Web UI 微服务将面对最终用户，所以它必须能够适应规模化的访问。我们将通过下述一些方法来实现高并发的 Web UI 微服务的设计。

2.4.1 使用 FeignClient 实现负载均衡调用

针对 Rest API 微服务对外提供的 Restful 的 HTTP 接口，我们将使用 Spring Cloud 工具套件中的 FeignClient 来进行访问，这个工具比起使用 JS 中的 Ajax (Asynchronous Javascript And XML) 或使用 RestTemplate 等工具都将更加高效和实用，而且使用更加简单。使用 FeignClient 通过声明式接口就可以实现 HTTP 调用，并且在调用的过程中，都将自动使用 Zuul 的动态路由和 Ribbon 的负载均衡服务。

2.4.2 使用 Hystrix 实现容错设计

在分布式环境中，当一个服务超载甚至失效时，如果这时对其继续调用，因为得不到回应，就会进入不断的等待状态之中。如果这是一种连锁调用，甚至会引起系统的雪崩效应。所以必须有一种机制来处理这种情况，以起到防患于未然的作用，这种机制就是



Hystrix 组件的断路器功能，它的作用相当于电路设计中的保护开关，当服务不可用或超载时，断路器切断服务，提供降级使用机制，当服务恢复正常后，断路器将自动接通服务。

我们将在 Web UI 微服务设计中，使用 Hystrix 组件提供的功能来实现断路器的设计，提供对服务访问的容错设计和降级使用机制。

2.4.3 使用非阻塞的异步编程技术实现高并发调用

Java 开发语言从 Java 8 之后，提供了一种非阻塞的异步的多线程的回调处理方法，它封装在 `CompletableFuture` 类中，使用这种编程方法的强大功能，就可以实现 Web UI 微服务的高并发设计。

2.4.4 使用分布式文件系统

在分布式系统中，最终用户使用的图片、视频等资源文件，必须使用一个独立的文件系统来管理。我们将使用轻量级的 FastDFS 来构建一个高可用的分布式文件系统，FastDFS 是一个开源的分布式文件系统，它不但容易使用和维护，也方便进行扩容。

FastDFS 由跟踪器（Tracker）和存储节点（Storage）两部分组成。

跟踪器管理所有的存储节点，并对存储节点的访问起到负载均衡调度的作用。通过配置多个跟踪器，可以构建一个高可用的文件系统。

存储节点用来存储文件，同时处理文件的存储、同步等管理工作。存储节点通常由多台服务器组成，每台服务器使用对等的设计，所以可以在任何时候进行增加或减少存储节点的数量。

为了支持大容量的文件存储，每个存储节点中使用了分组（或分卷）的设计，每个分组可以存储在不同的服务器上。

所以，使用 FastDFS 可为我们提供一个高可用和高性能的海量文件系统。

2.5 微服务之间调用规则设计

为了保证各个微服务的独立性，并减少通信的复杂性，提高微服务之间的调用效率，我们对微服务之间的调用，做出了如下几种约定。



1. 通过 Web UI 调用 Rest API

服务之间的调用，主要是指 Web UI 微服务对 Rest API 微服务的调用。一个 Web UI 在一次调用之中，可以同时调用多个 Rest API，并且这种调用将通过高并发设计来实现，即多个调用将会由不同的线程所执行，所以即使是多个调用，也不会影响程序的执行效率。

2. Rest API 之间只能通过 MQ 进行互相通信

为了保证 Rest API 的高性能特性，同时避免因为一个 Rest API 的功能变更对其他 Rest API 的影响，所以禁止在 Rest API 之间进行相互调用。Rest API 之间如果需要进行通信，可以使用异步消息来实现，即通过消息总线实现异步通信。

3. Web UI 之间可使用与之对应的实例进行相互跳转

Web UI 之间不进行相互调用，但可以进行相互跳转，这种跳转不是直接使用地址栏的链接来进行的，而是通过注册实例实现跳转，当一个注册实例有多个副本时，将可进行负载均衡管理。

2.6 数据最终一致性设计

集中式的数据管理可以在一个事务中完成，所以能保证数据的高一致性。对于微服务的多服务架构，数据将由不同的微服务进行分散管理，所以要保证数据的一致性，必须有合理的设计。

假如我们有商品和订单两个微服务，那么在订单服务中生成订单时，将需要在商品服务中进行库存减少的操作，而在订单服务中进行订单查询时，将有可能需要在商品服务中查询订单相关的商品信息，这就涉及不同微服务中数据一致性的问题。

我们可以依据 CAP 原理的 BASE 理论来实现数据最终一致性设计。

CAP (Consistency, Availability, Partition tolerance) 即一致性、可用性、分区容错性三者不可兼得。

BASE (Basically Available, Soft state, Eventually consistent) 即基本可用、软状态、最终一致性。BASE 是对 CAP 中一致性和可用性进行权衡的结果。

数据最终一致性设计具体的实现主要由两种类型的操作完成：一种是通过调用各个 Rest API 实现实时同步操作；另一种是使用消息通道以事件响应的方式进行异步处理。



比如在商品服务和订单服务的例子中，当用户下单时，即同时调用两个服务的 Rest API，一方面，通过订单服务的 Rest API 生成订单，另一方面通过商品服务的 Rest API 更新库存数量。而当订单状态修改时，即由订单服务的 Rest API 发布一个异步消息，商品服务的 Rest API 通过订阅消息来执行相应的更新操作。

2.7 分布式集群架构设计

通过微服务的治理环境，使用多副本发布的每个微服务，最终都将被自动纳入到微服务的负载均衡管理体系之中，这些微服务应用与我们搭建的数据库集群、分布式文件系统

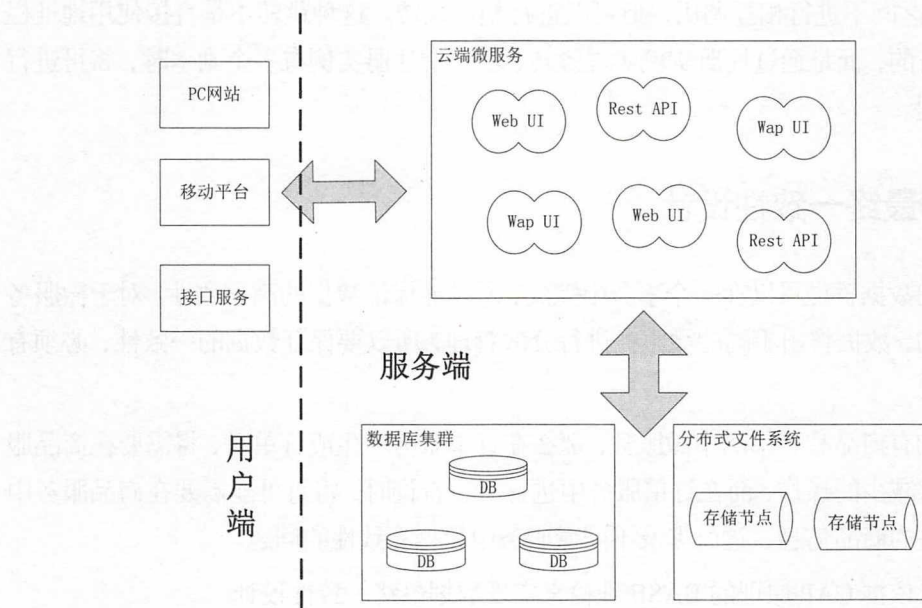


图 2-2 微服务分布式集群体系结构

这个集群体系将由很多不同的服务器所组成，这些服务器既可以分布在同一个局域网之中，也可以进行跨区域、跨机房分布，组成一个庞大的分布式体系结构。总之，使用分布式集群架构设计，可以搭建一个稳定可靠并可持续扩展的系统平台。

2.8 微服务运行环境安全设计

系统的安全设计包括防火墙设计、防攻击设计、访问控制设计、数据保密设计等各个方面的内容。而防火墙设计是系统安全的第一道屏障，我们将使用防火墙为微服务架构的服务器组建提供一个安全可靠的分布式环境，如图 2-3 所示。

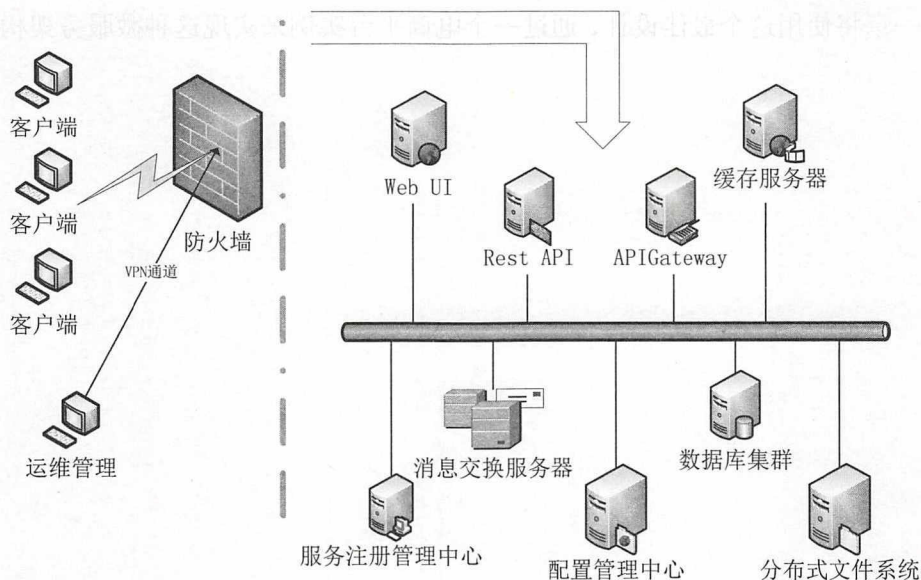


图 2-3 微服务运行环境安全设计网络拓扑结构

使用防火墙，我们可以组建一个局域网环境，在这个局域网环境之中，各种服务器的组建将会更加容易，服务器的配置会更加简单，服务之间的通信也会更加方便和快速。

使用防火墙，我们还可以将不同区域、不同机房的服务器通过 VPN 连接起来，提供一种更加安全的访问方式。

2.9 小结

本章介绍了怎么围绕业务功能来进行合理地划分微服务的方法，并在此基础上创建了两种类型的微服务，即专门用来存取数据的 Rest API 微服务和主要实现前端设计的 Web UI 微服务，使用这两种类型的微服务，通过微服务治理，并结合使用数据库集群设计、缓存

设计、分布式文件系统设计、微服务的集群设计、安全的分布式环境设计等一系列分布式的架构设计，就可以构建成一个微服务架构的最佳设计，并在微服务架构本身所体现出来的优势和缺点之间进行权衡与调整，最终充分发挥微服务架构的绝对优势。

其中，有关微服务的集成测试和部署的复杂性问题，我们将在第三部分的运维篇中介绍使用自动化构建工具来解决。

下一章将使用这个最佳设计，通过一个电商平台实例来实现这种微服务架构设计。



3

电商平台微服务设计实例

上一章介绍了微服务架构的最佳设计方法,本章将使用这种方法通过一个电商平台实例来实现微服务架构设计。电商平台是一个大众化的应用平台,大家对它的功能都比较熟悉,我们将通过电商平台的微服务设计,体会微服务在实际生产中的具体使用方法。

3.1 电商平台总体设计

电商平台是电子商务交易平台的简称,是指通过互联网为企业和个人提供网上交易的管理平台。电商平台可提供一个网上自由交易的场所,为普通用户(顾客)和虚拟商铺(商家)建立一种可信的买卖关系,通过互联网实现不受地域和时间等条件所限制的贸易行为。

我们将以一个通用的电商平台为例进行设计,但不做太多复杂的功能,也不关注太多的细节变化,我们只要求大体上能完成一个网上购物的流程,以此来体会微服务架构在实际应用中的使用方法。

3.1.1 总体业务流程设计

如图 3-1 所示是电商平台的一个总体业务流程设计。

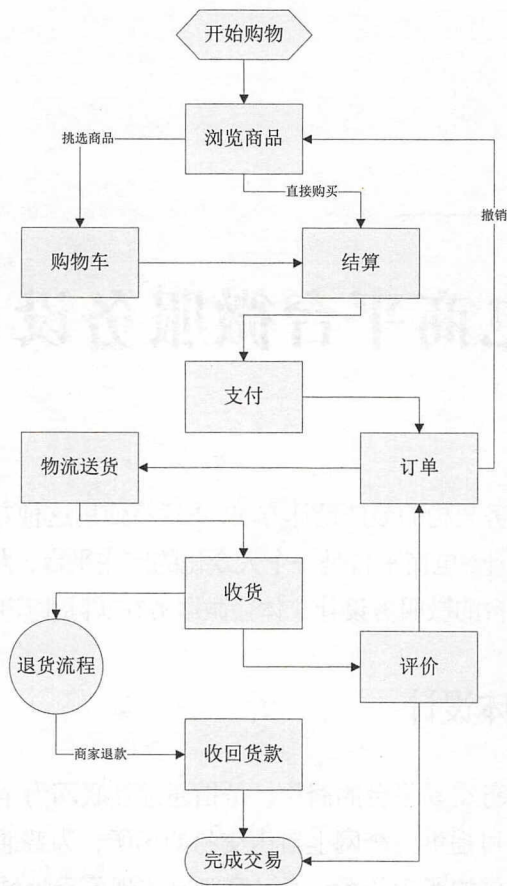


图 3-1 电商平台基本业务流程图

这个流程表示，顾客进行网上购物时将从浏览商品、挑选商品开始，然后经过结算、支付的过程，生成了一个交易的订单。商家通过后台的订单管理就可以确认顾客的交易行为，并联系物流公司进行发货处理。最后，顾客收到商品后即完成了一个正常的交易流程。顾客还可以对这次交易进行评价。

其中，在结算的过程中，如果顾客感到不满意，可以在一定的有效期内对生成的订单执行撤销交易的操作。

另外，如果顾客收到货物后对商品质量感到有问题，还可以申请售后服务或者直接提出退货，开启退货的申请流程。商家审核退货后，可以给顾客退回货款，从而结束一个交易。

要完成一个完整的交易过程,还将涉及其他业务流程的设计,这些流程包括如下几个方面的设计。

- 顾客在购买商品之前,必须先进行平台用户的注册,然后执行个人基本资料的编辑,新增和维护收货地址等操作。平台顾客也可以注册为商家的会员,然后由商家提供会员级别的服务。
- 商家将通过后台管理执行商品发布、订单管理、物流处理、退货审核、会员注册及其管理等基本操作流程。
- 平台将对商家入驻有一个管理流程,包括商家的管理、商家注册与审核、商家权限管理等功能。

3.1.2 总体业务功能设计

电商平台总体业务功能的设计包括如下几个方面。

- 涉及商品展示方面的功能,将包括商品的类目配置及其管理、商品的库存、定价、商品信息编辑、商品上下架管理等内容。
- 在进行交易操作方面,将包括顾客管理、会员管理、购物车管理、支付管理、订单管理、物流管理等方面的内容。
- 在商家的管理方面,将包括商家的入驻与注册的审核、商家的操作权限配置、商家的账户管理、结账和对账等方面的内容。

上述这些功能,根据其面向的用户对象不同,可以将电商平台的总体业务功能分为:面向顾客的门户商城、面向商家的管理后台和面向平台运营方的管理后台三大部分,如图 3-2 所示。



图 3-2 电商平台总体业务功能结构图

其中,门户商城就相当于商家的店铺,是商家展示商品、顾客浏览商品并进行现场交



易的地方。商家管理后台是商家进行商城事务日常管理的操作平台。平台管理后台是平台运营方的一个管理后台，它是用来管理商家及其操作权限的平台。

3.2 电商平台业务模型设计

根据电商平台的总体业务功能结构，可以创建相应的业务模型。其中，对于门户商城，我们将只提供移动商城的业务模型设计。移动商城可以适用于手机、iPad 等移动设备的访问。移动设备可以通过普通浏览器、App、微信公众号或小程序等方式来访问移动商城。

所以，电商平台最终的业务模型的设计包括：移动商城业务模型、商家后台管理模型、平台后台管理业务模型三大业务功能模型的设计。

3.2.1 移动商城业务模型

移动商城的业务功能包括：商品展示，分类查询，购物车管理，结算下单，订单查看及评价，物流查询及跟踪，个人信息管理等。它的业务模型如图 3-3 所示。

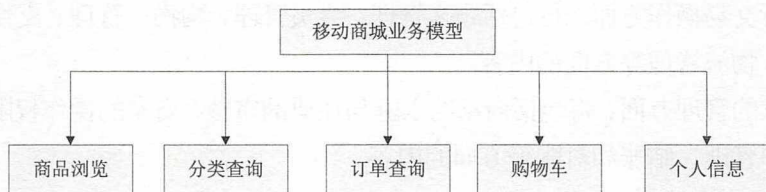


图 3-3 移动商城业务模型

其中，各个模块的功能简单介绍如下：

- 商品浏览提供商品搜索和查询等功能，包括商品列表分页展示和单个商品详情查看等，并在商品详情查看中提供购买下单的功能。
- 分类查询提供按分类列表查询商品的功能。
- 订单查询可以实时显示订单状态，查询订单的物流进度，并可以进行收货确认和对完成交易的订单进行评价等。
- 购物车提供增、删、改、查全面的管理功能，方便顾客执行加入商品、移除商品、更改一个商品的购买数量等操作。
- 个人信息包括顾客的基本信息的管理，比如手机号、联系人、收货地址等信息的管

理。同时，顾客也可以注册为某一商家的会员，注册会员后，将可以享受商家提供的优惠、折扣和积分等会员特权服务。

3.2.2 商家管理后台业务模型

商家管理后台的业务功能包括：商家的用户管理、商品管理、商家的支付账户设置、订单处理及查询、物流管理、会员管理及顾客浏览商品的点击率统计等功能。商家管理后台的业务模型如图 3-4 所示。

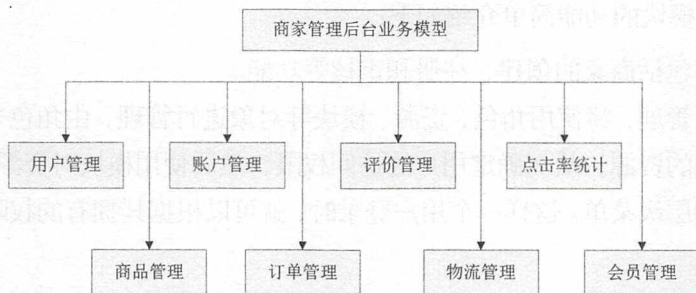


图 3-4 商家管理后台业务模型

其中，各个模块的功能简单介绍如下：

- 用户管理为商家提供了管理后台操作用户的功能，可以增加和删除用户，并为每一个用户配置操作权限。
- 商品管理可以进行商品添加、编辑及商品的上下架等操作。
- 订单管理可为商家提供订单处理、查询和订单统计等功能。
- 物流管理可为商家提供订单发货和物流管理等功能。
- 支付管理包括商家的收款账户设置，收款记录查询和统计等功能。
- 评价管理给商家提供了查看顾客对订单评价的功能。
- 点击率统计可对顾客浏览商品的行为进行查询和统计。
- 会员管理为商家提供了会员等级和相关特权设置，并对所属的会员进行集中查询和管理等功能。

3.2.3 平台管理后台业务模型

平台管理后台包括平台操作员的权限配置和管理，商家注册和审核，商家用户的权限

配置和管理等功能。平台管理后台的业务模型如图 3-5 所示。

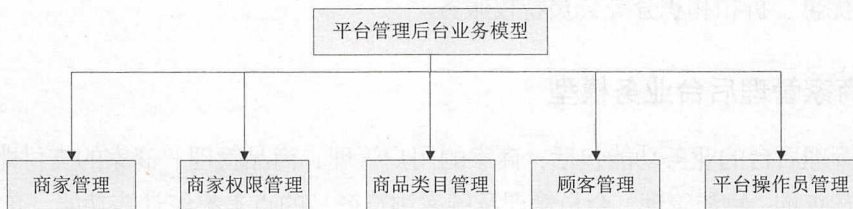


图 3-5 平台管理后台业务模型

其中，各个模块的功能简单介绍如下：

- 商家管理包括商家的创建、注册和审核等功能。
- 商家权限管理，将使用角色、资源、模块等对象进行管理，由角色来决定一个用户可以访问的资源，从而确定用户的访问权限，然后使用模块对资源进行层级管理，形成一种层级菜单。这样一个用户登录时，就可以根据其拥有的权限分配合理的菜单结构。
- 商品类目将由平台进行统一管理，不提供给商家操作这一方面的功能。平台将按合理的标准提供全面的分类体系。
- 顾客管理包括顾客的注册和个人信息编辑等功能，将由平台方进行统一管理。同时，顾客也可以注册为商家的会员。
- 平台操作员管理提供了平台操作员创建和权限管理等功能，通过操作员管理可以实现平台访问控制的安全设计。

3.3 创建 Rest API 微服务

根据电商平台的总体设计及其各个业务模型的功能，就可以开始创建或划分微服务。

我们将使用第 2 章中介绍的微服务架构的设计方法来划分微服务，即使用水平划分方法和垂直划分方法来创建微服务。

首先使用水平划分方法，按电商平台的业务功能，进行 Rest API 微服务划分，划分出来的结果包括如下一些服务：

- 类目服务。
- 商品服务。



- 购物车服务。
- 订单服务。
- 支付服务。
- 物流服务。
- 评价服务。
- 顾客服务。
- 会员服务。
- 点击率服务。
- 商家服务。

通过上述这些服务,就可以创建相关的 Rest API 微服务。Rest API 微服务是一个独立的小应用,并且都有独立的数据库,可以独立部署和独立运行。Rest API 微服务使用实体对象进行数据的存取操作,然后对外提供基于 HTTP 协议的 Restful 接口服务。

这些微服务大体上的功能说明如下:

- 类目服务,使用二级分类体系,对外提供分类信息的录入、查询、修改和删除等各项操作的接口服务。在应用层面上,可以根据不同的业务需求使用不同的接口服务,在商城和商家后台管理中,只提供分类的信息查询,而在平台管理后台中,可对分类进行编辑和管理。
- 商品服务,提供了商品介绍、商品编辑管理、商品上下架等操作的接口服务。这些服务可用于商家管理后台中,为商家提供商品管理的功能。在商城应用中只需要使用商品查询、搜索、商品详情显示等功能接口。
- 购物车服务,主要为商城的顾客在选购商品时提供了方便,同时购物车服务也提供了对选购商品进行加减、移除等相关的接口服务。
- 订单服务,在商城中为顾客提供订单生成、订单查询和评价等功能接口,在商家管理后台中可为商家提供订单管理、查询和统计等功能接口。
- 支付服务,在商城中提供结算支付的服务,在平台管理中提供了服务费计算和利润结算的功能接口,同时也可可为商家提供收款查询和对账等功能接口。
- 物流服务,在商城中为顾客提供物流跟踪及收货确认等服务接口,在商家后台中提供了发货处理和查询统计等功能接口。
- 评价服务,在商城中为顾客交易完成后提供了订单评价的功能,同时也为顾客在选



购商品时提供了商品评价及其查询的功能，评价可为顾客购物提供参考。

- 顾客服务，顾客是平台的用户，顾客服务提供了用户的注册及其个人信息管理等功能，可以对外提供包括注册、登录、个人信息编辑、收货地址管理等功能接口。
- 会员服务，会员是商家的用户，顾客在购物过程中可以在任何一个商家中注册成为会员，注册会员后，将可享受商家提供的特权服务，比如购物折扣、会员积分等，会员服务将为相关功能的实现提供接口服务。
- 点击率服务，点击率是记录顾客浏览商品的足迹，这些数据可为商家的销售提供决策参考。点击率服务为商家提供查询和统计的功能接口。
- 商家服务，可提供商家创建、编辑和权限管理等接口服务，可为平台管理后台实现商家注册、审核和商家用户的权限管理等功能。

3.4 创建 Web UI 微服务

创建了 Rest API 微服务之后，就可以使用垂直划分方法，根据每个 Rest API 微服务来实现前后端分离设计，创建 Web UI 微服务。

根据电商平台的业务模型设计，我们将分别从移动商城、商家管理后台和平台管理后台三个方面来创建 Web UI 微服务。

3.4.1 移动商城 Web UI 微服务

移动商城的功能包括商品的查询、搜索、分类查询、顾客购物下单、购物车管理、订单查询、物流跟踪查询、个人信息和会员卡管理等功能。

根据移动商城的业务模型设计，可以创建出下列的 Web UI 微服务：

- 分类查询。
- 商品查询。
- 购物车管理。
- 订单查询。
- 物流跟踪。
- 个人信息管理。
- 会员卡管理。



这些微服务应用的设计将面向移动端设备，与 PC 端应用的设计会有一些差别。因为移动端操作界面较小，所以我们主要使用单页 H5 来设计页面视图，这种单页 H5 的适用范围比较广泛，不但可以使用普通浏览器来访问，也可以适用于 App 程序的调用。

3.4.2 商家管理后台的 Web UI 微服务

商家管理后台的功能，包括商家用户管理、商品管理、订单管理、物流管理和会员管理等方面的功能。这里的每一项功能，将分别由一个单独的微服务应用所提供。

商家管理后台的 Web UI 微服务，包括如下一些应用：

- 用户管理。
- 商品管理。
- 订单管理。
- 物流管理。
- 评价查询。
- 账户管理。
- 会员管理。
- 点击率统计。

商家管理后台将实现安全的访问控制设计，而商家管理后台的功能又将由不同的应用所提供，所以，为了统一用户登录，提供友好的用户体验，我们还将使用一个 SSO（单点登录）设计。

SSO 也是一个独立的微服务应用，一方面提供统一的访问控制功能，另一方面提供统一的认证和授权管理的功能，即不管用户在哪一个应用之中登录之后，就可以获得访问其他应用的授权。

这样，当用户在使用管理后台的不同功能时，将感觉不到在不同应用之间的跳转，而会感觉像在使用一个应用系统的不同功能模块一样。

3.4.3 平台管理后台 Web UI 微服务

平台管理后台也是一个独立的 Web UI 微服务应用，它将通过调用商家服务实现商家注册和审核、商家权限配置等管理功能。



平台管理后台的操作对象为平台运营方，使用范围比较小，所以将使用比较简单的设计方法，即使用一个单独应用来完成如下所示的相关管理功能。

- 本地用户管理。
- 商家管理。
- 商家权限管理。

另外，平台管理后台的访问控制设计，也将使用比较简单的方法来实现。

3.5 电商平台微服务体系结构

经过上面一系列的微服务设计，现在，我们可以使用一个思维导图来表示这个完整的电商平台的微服务架构设计模型，如图 3-6 所示。

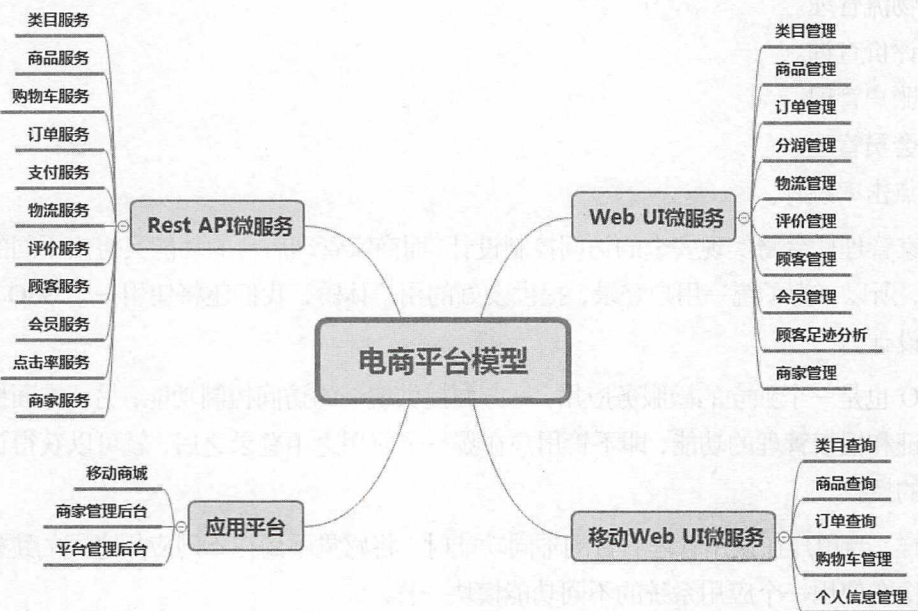


图 3-6 电商平台微服务设计模型

这是一个比较简单的电商平台微服务设计，并且我们也使用了粗粒度的微服务划分方法来划分微服务，但是这已经可以分出二十多个微服务了，如果我们再结合使用多副本的方式进行部署，那么一个电商平台至少将要运行四十多个微服务。

另外,有关高性能的 Rest API 微服务设计和高并发的 Web UI 微服务设计,以及微服务之间的调用关系等设计,与第2章中的设计类似,不再赘述。

3.6 小结

本章设计了一个简单的电商平台,并使用这个电商平台来进行微服务设计,首先按电商平台的业务功能创建了 Rest API 微服务,然后在此基础上,根据电商平台的业务模型,使用 Web UI 微服务组建了移动商城、商家管理后台和平台管理后台等应用平台。

在后面的开发实例中,我们将根据这些微服务的设计来构建项目工程。



第二部分

开发篇

第4章 开发工具选用及 Spring Boot 基础

第5章 电商平台微服务工程设计

第6章 微服务治理基础服务开发

第7章 Rest API 微服务开发

第8章 Web UI 微服务开发

第9章 电商平台移动商城开发

第10章 商家管理后台与 SSO 设计

第11章 平台管理后台开发

这一部分以一个电商平台为例,进行了微服务的开发。在开发的整个过程中,重点阐述了高性能的 Rest API 微服务和高并发的 Web UI 微服务的实现方法,并通过类目服务、商品服务、订单服务、商家服务等项目工程实例,开发了移动商城、商家管理后台和平台管理后台等微服务应用平台。



4

开发工具选用及 Spring Boot 基础

因为 Spring Cloud 是以 Spring Boot 框架为基础进行开发的，所以在使用 Spring Cloud 工具套件之前，我们需要了解一下 Spring Boot 框架的使用方法，同时这里也说明一下开发工具的选择和开发环境的配置，为后面的微服务开发做好充分的准备。

Spring Boot 是基于 Spring 开发框架而开发的一个全新的框架，除了具有 Spring 的特点，还具有如下一些显著的特点：

- 为所有 Spring 开发提供一个更快、更广泛的入门体验。
- 具有开箱即用的默认配置功能，能根据项目依赖进行自动配置。
- 提供大型项目（例如嵌入式服务器、安全性、指标、运行状况检查和外部配置）通用的一系列非功能性功能。
- 绝对不会生成代码，也不需要 XML 配置。

也就是说，使用 Spring Boot 框架，将没有复杂的配置，一些组件的引用大都可以使用默认的自动配置，并且在应用程序运行和打包中还可以嵌入 Tomcat、Jetty 等服务器，所以开发将更加简单和快速，而部署也将更加容易。



4.1 开发工具选择

使用 Spring Boot 开发框架，虽然有很多开发工具可以选择，比如 Eclipse、NetBean、Interllij IDEA 等，但本书还是极力推荐大家使用 Interllij IDEA（简称 IDEA），本书的实例也是使用这一工具开发的。IDEA 不仅在智能代码助手、工程管理、版本控制等各个方面都很优秀，而且已经包含了一些常用的工具插件，不用再费力寻找和安装，并且对 Spring Boot 的开发更是提供了全面而独到的支持。

IDEA 可以从其官方网站中下载、安装和使用：

<http://www.jetbrains.com/idea/>

IDEA 全面支持 Spring、Spring Boot、Java EE、Android、JavaScript、HTML/CSS、Node.js 等项目的开发。

4.2 开发环境配置

开发环境主要是安装 JDK、Maven 和 Git 客户端。

JDK 需要使用 1.8 或以上的版本，根据你使用的操作系统，选择下载安装包进行安装，网址如下：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

IDEA 已经包含 Maven 项目管理工具，以及 CVS（Concurrent Versions System）、Subversion、Git 等版本控制管理工具。本书的实例都存放于 Git 代码仓库之中，为了使用 Git 的功能，还需要下载一个客户端。Git 客户端可以从其官网上根据你使用的操作系统选择合适的版本下载和安装：

<https://git-scm.com/downloads>

安装完成后在 IDEA 中配置 Git 的执行路径即可使用。如图 4-1 所示为 Mac OS 环境的配置实例，其中的路径配置“/usr/local/bin/git”即为 Git 的安装路径和执行程序。如果配置正确，单击“Test”按钮即可返回执行成功的提示和 Git 的版本号。



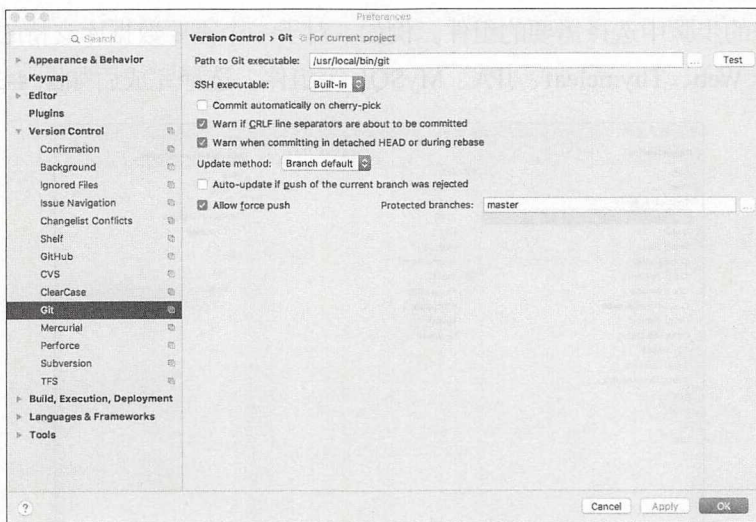


图 4-1 IDEA 的 Git 配置

4.3 创建 Spring Boot 工程

开发环境配置完成之后，即可使用 IDEA 进行开发。使用 IDEA 来开发 Spring Boot 项目是非常容易的，现在我们来创建一个简单的 Spring Boot 项目工程，在该项目工程中包含了简单的数据管理和视图设计的功能。

首先，新建一个项目，并选择使用 Spring Initializr 来生成一个 Spring Boot 项目，如图 4-2 所示。在 Project SDK 中选择安装的 JDK1.8。

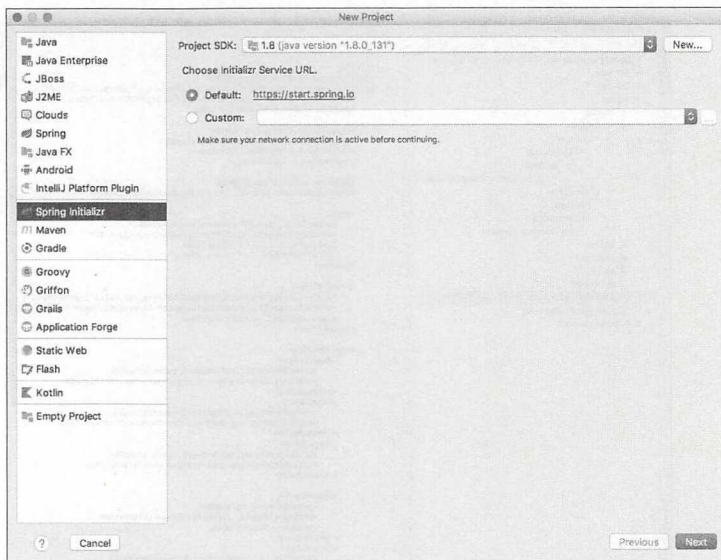


图 4-2 使用 Spring Initializr 创建 Spring Boot 项目

在接下来的步骤中选择需要的组件。例如，对于一个需要数据库支持的 Web 网站来说，可以选择 Web、Thymeleaf、JPA、MySQL 等组件，选择完成后如图 4-3 所示。

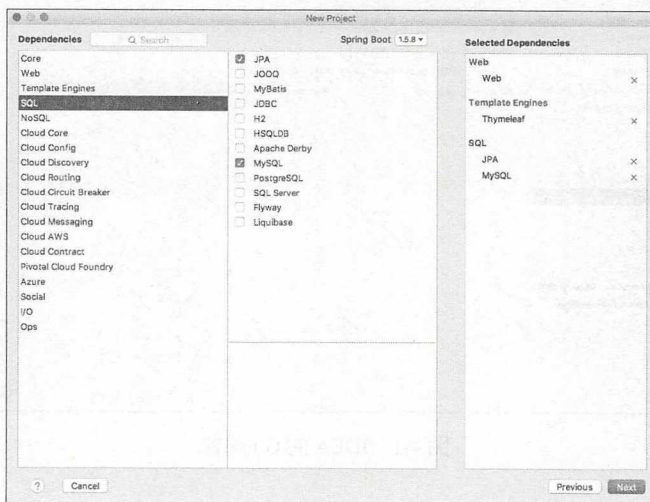


图 4-3 选择项目中使用的组件

继续下一步，选择项目文件存放的路径，完成后将生成一个项目的文件结构，如图 4-4 所示，其中主要包含如下各项内容。

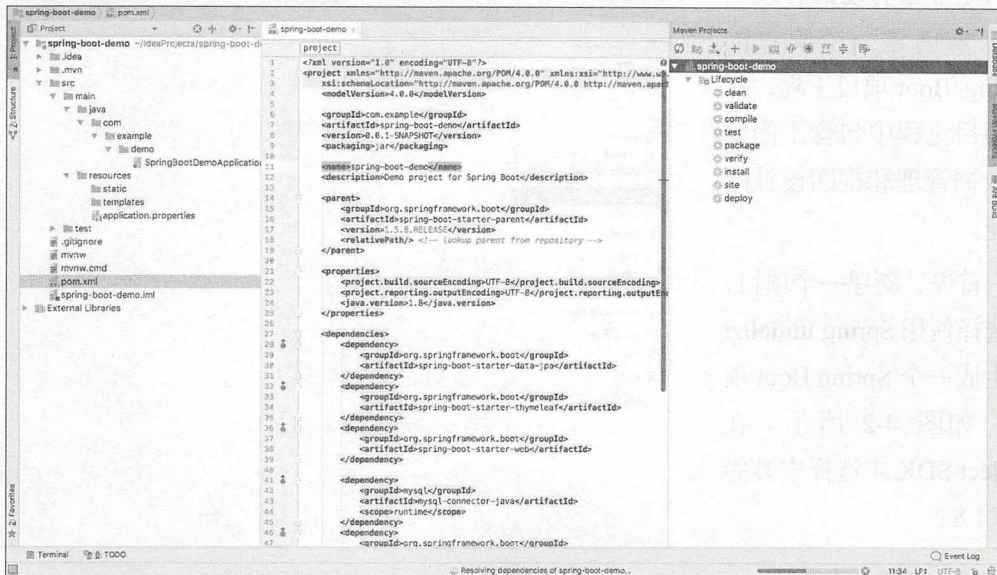


图 4-4 生成的 Spring Boot 项目

- 项目对象模型：pom.xml。
- 启动引导程序：SpringBootDemoApplication.java。
- 配置文件：application.properties。

如果在图 4-4 中，打开右侧的 **Maven Projects** 没有看到项目工程，可以将工程根目录的 pom.xml 文件拖拉过去，就可以使用 **Maven** 进行项目管理了。打开 pom.xml 文件，可以看到我们在上面的生成步骤中选择的几个组件的依赖引用。

pom.xml 完整的内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>spring-boot-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/><!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.
outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
```




```

        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>

```

在生成项目的过程中，将会下载一些引用的依赖包，等待这些依赖包加载完成之后，即可执行编译或打包等操作。

现在，在右侧的 **Maven Projects** 中，双击 **compile** 将对这个工程进行编译。执行过程若输出类似如下所示的信息，就表明编译成功了。

```

[INFO] -----
[INFO] Building spring-boot-demo 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ spring-
boot-demo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.

```



```
[INFO] Copying 1 resource
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ spring-
boot-demo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source files to /Users/spring-boot-demo/target/classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

从上面的编译结果中可以看出，现在项目中只有一个程序通过了编译，这个程序就是项目的启动程序，它的代码很简单，使用了一个注解“@SpringBootApplication”，表示这是一个 Spring Boot 应用的引导程序，并且这个程序只有一个“main”方法，如下所示：

```
@SpringBootApplication
public class SpringBootDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootDemoApplication.class, args);
    }
}
```

但是，现在还不能运行工程，因为我们使用了 MySQL，还没有配置数据源，如果这时运行项目程序将会报错。下面来做一些配置并添加一些功能。

其中，生成的配置文件 `application.properties` 现在还是一个空白的文件，在后面我们将根据需要增加一些配置信息。Spring Boot 使用的组件一般都具有开箱即用的功能，即有一些默认的自动配置。例如我们使用的 Web 应用，默认使用的端口为“8080”，而使用的 Thymeleaf 模板，默认的页面文件存放路径为“templates”等。

4.4 使用 JPA

JPA (Java Persistence API) 是 Java 的持久层规范接口，使用 JPA 能方便我们直接针对领域对象来设计业务功能，而不用去关注表结构设计。当我们设计好实体之后，通过继承 JPA 的存储库接口，就能实现实体的持久化。程序运行时，JPA 将根据实体定义自动生成和更新表结构。所以，使用 JPA 也能很好地适应快速迭代所引起的表结构变化，非常适合用于敏捷开发方法。JPA 的实现使用了 Hibernate 框架，所以它的一些设计与使用 Hibernate 是差不多的。

下面将通过一个简单的实体设计来说明使用 JPA 的方法。



4.4.1 数据源配置

使用数据库，首先必须进行数据源的配置。假设已经在本地机器上安装了 MySQL，如果你的 MySQL 中没有数据库 “test”，请按照下列方法创建一个：

```
CREATE DATABASE IF NOT EXISTS test DEFAULT CHARSET utf8 COLLATE utf8_general_ci;
```

注意这里使用了字符集 “utf8”，它适合中文环境的开发。然后使用下列语句创建一个具有完整权限的数据库用户，这个语句假设数据库就在你的本地机器上，如果你的数据库在局域网上，可以将其中的 “localhost” 换成你机器的 IP 地址。

```
grant all privileges on test.* to 'root'@'localhost' identified by '12345678' with grant option;
```

有了数据库的操作权限，我们就可以在工程的配置文件中配置数据源，如下所示：

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test?characterEncoding=utf8&useSSL=false
spring.datasource.username=root
spring.datasource.password=12345678
```

4.4.2 JPA 配置

在 Spring Boot 中使用 JPA 将不再使用复杂的 XML 文件配置，我们只需在通用的配置文件中设定 JPA 一些运行参数，以及使用一个简单的配置类设置好实体工厂和存储库就可以了。

下面是使用 MySQL 数据库的一个 JPA 配置，配置中我们将 “ddl-auto” 设置为 “update”，表示将根据实体设计在程序运行时自动更新表结构。

```
spring.jpa.show-sql=true
spring.jpa.database=MYSQL
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

为了让 JPA 能找到我们自定义的实体类和存储库接口设计，我们还需要在程序中增加一个 JPA 配置类，配置类的实现代码如下所示：

```
@Configuration
```




```

@EnableTransactionManagement(proxyTargetClass = true)
@EnableJpaRepositories(basePackages = "com.**.repository")
@EntityScan(basePackages = "com.**.domain")
public class JpaConfiguration {
    @Bean
    PersistenceExceptionTranslationPostProcessor
    persistenceExceptionTranslationPostProcessor(){
        return new PersistenceExceptionTranslationPostProcessor();
    }
}

```

其中，注解“@EntityScan”设定了实体类的扫描路径。注解“@EnableJpaRepositories”设定了存储库接口的存储位置。

4.4.3 数据实体设计

这里使用一个非常简单的数据对象——国家代码来演示数据实体的设计方法。国家代码对象只具有代码和名称两个属性，它的实体设计如下所示：

```

@Entity
@Table(name = "country")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String countryname;
    private String countrycode;
    .....
}

```

其中，注解“@Table”将在数据库中生成指定名称的数据表。

4.4.4 存储库接口设计

设计了实体之后，我们只要为这个实体创建一个存储库接口，就可实现实体的持久化。国家代码的存储库接口的实现代码如下所示：

```

@Repository
public interface CountryRepository extends JpaRepository<Country, Long>{
}

```

这个接口继承了 JPA 的存储库“JpaRepository”，并使用实体“Country”作为初始化对象。



4.4.5 单元测试

到目前为止，我们并没有写一行执行数据存取操作的代码，例如写一些存取数据的 SQL 语句之类的代码，但是我们现在已经可以对数据库进行一般的数据存取操作了。

下面通过一个测试来验证一下怎么样通过实体对象实现对数据库的存取操作。

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ContextConfiguration(classes = {JpaConfiguration.class, SpringBootDemoApplication.class})
public class SpringBootDemoApplicationTests {
    private static Logger logger = LoggerFactory.getLogger(SpringBootDemoApplicationTests.class);

    @Autowired
    private CountryRepository countryRepository;

    @Before
    public void createCountry(){
        Country country = new Country();
        country.setCountrycode("86");
        country.setCountryname("中国");
        countryRepository.save(country);
        assert country.getId() > 0 : "create error";
    }

    @Test
    public void getData() {
        List<Country> countrys = countryRepository.findAll();
        assert countrys != null : "getdata is null";
        for(Country country : countrys) {
            logger.info("==== country name = {}", country.getCountryname());
        }
    }
}
```

第一次运行这个测试用例，如果全部通过，即可完成如下一些功能的验证：

- 在 test 数据库中创建了一个国家代码表 “country”。
- 在表中插入了一条记录。
- 通过查询返回一个国家代码列表。

测试完成之后，我们将可以在控制台中看到如下所示的输出记录：

```
==== country name = 中国
```



4.5 使用 Thymeleaf

Thymeleaf 是 Spring Boot 开发框架中前端设计的一个通用模板，它可以直接使用 HTML 页面文件，并支持 XML、HTML5 设计，完全覆盖并替换了 Java 开发框架中通常使用的 JSP。在高版本的 Spring Boot 中将不再提供对 JSP 的支持。Thymeleaf 也有类似于 LSTL 的标签库和功能丰富的工具函数。

Thymeleaf 的配置我们将使用默认的自动配置。有关这些配置参数的设置也可以从 Spring Boot 的自动配置源程序看出来，它用几个静态变量定义了 Thymeleaf 的默认配置参数，如下所示：

```
package org.springframework.boot.autoconfigure.thymeleaf;
.....
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING = Charset.forName("UTF-8");
    private static final MimeType DEFAULT_CONTENT_TYPE = MimeType.valueOf(
("text/html"));
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
    .....
}
```

如果我们在项目中没有指定配置参数时，就将默认使用这些自动配置参数，这些默认的配置（这里使用了 YAML 格式）如下所示：

```
spring:
  thymeleaf:
    prefix: /templates/
    suffix: .html
    encoding: UTF-8
    content-type: text/html
```

4.5.1 控制器设计

在这个例子中将使用一个简单的控制器，通过定义一个简单的 URL “/” 来返回一个查询数据的页面“index”。数据查询使用了上面定义的存储库接口“CountryRepository”，通过调用其 findAll()方法取得列表数据。控制器的设计如下所示：

```
@Controller
public class CountryController {
    @Autowired
```




```

private CountryRepository countryRepository;

@RequestMapping(value="/")
public String test(ModelMap model) {
    List<Country> list = countryRepository.findAll();
    model.addAttribute("country", list);
    return "index";
}
}

```

4.5.2 视图设计

我们使用一个简单的页面设计来显示数据的输出列表，实现的代码如下所示：

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>国家代码</title>
</head>
<body>
    <div>
        <ul th:each="country:${country}">
            <li>
                <p th:text="'代码: '+${country.countrycode}"></p>
                <p th:text="'国家: '+${country.countryname}"></p>
            </li>
        </ul>
    </div>
</body>
</html>

```

其中，我们使用标签语言“th”在“”控件中通过一个循环语句“th:each”输出了国家代码的列表，并使用列表控件“”来显示数据。

4.6 运行与部署

经过了上面一些简单的配置与开发，现在已经可以运行这个项目工程了。

上面实例的完整代码可以从下列链接通过 IDEA 进行 Checkout 或直接在浏览器中下载：

<https://gitee.com/chenshaojian/spring-boot-demo.git>



Spring Boot 项目默认已经内嵌 Tomcat 组件, 所以只要运行 `SpringBootApplication` 程序即可启动应用进行调试。

如图 4-5 所示为在 IDEA 中运行应用之后, 控制台的输出情况。

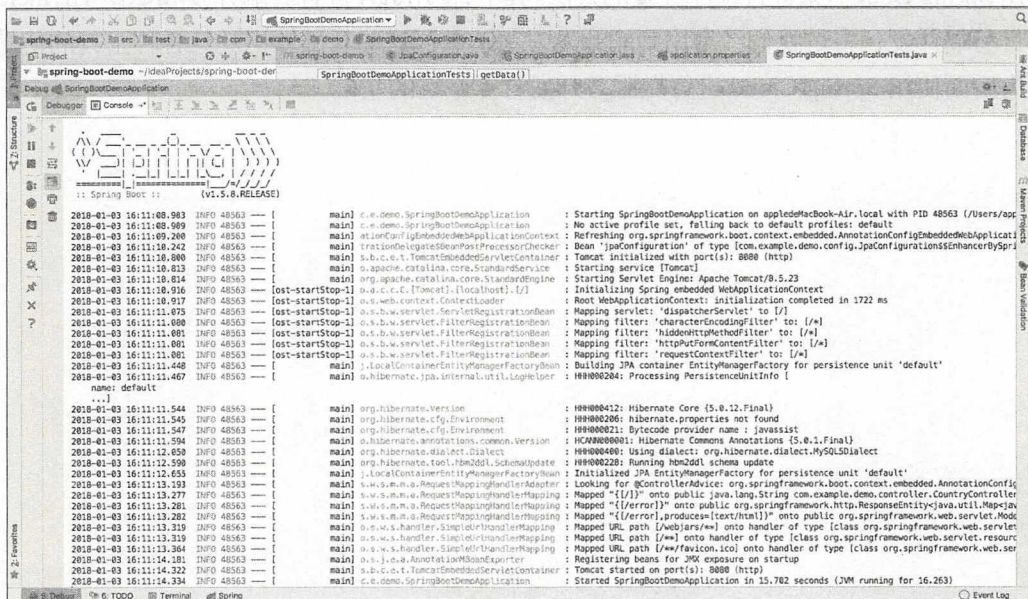


图 4-5 项目运行控制台

启动后通过浏览器打开如下链接:

<http://localhost:8080>

运行效果如图 4-6 所示。

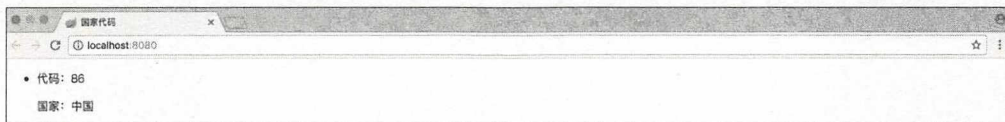


图 4-6 实例运行效果

如果将工程打包之后, 可以通过 jar 包在命令终端中使用如下方式运行:

```
java -jar spring-boot-demo-0.0.1-SNAPSHOT.jar
```


4.7 小结

本章简要介绍了开发工具的选择和开发环境的安装配置,并使用一个简单实例演示了使用 Spring Boot 框架进行程序开发的方法。其中,使用 JPA 进行后端开发和使用 Thymeleaf 进行前端开发,将是我们经常用到的开发方法,掌握这些基础知识及其各种工具的使用,将为后面使用 Spring Cloud 进行微服务开发打下坚实的基础。



5

电商平台微服务工程设计

在第3章的电商平台微服务设计实例中,我们使用粗粒度的微服务划分方法,将电商平台的业务模型划分出二十多个微服务。那么,怎么来组织项目工程开发这些微服务呢?是不是每一个微服务都建立一个对应的项目呢?当然不是,那样的话,将会有很多的项目工程。

本章就以电商平台微服务架构设计实例来说明如何组建工程和进行相关的微服务开发。

5.1 微服务工程结构

微服务的工程组建,我们将按业务类型来进行组织,即将同一业务类型的微服务放在同一个项目工程之中。同一业务类型的微服务包括 Rest API 微服务和 Web UI 微服务,其中 Rest API 微服务又包含了业务领域的数据库建模和接口服务等方面的设计,而 Web UI 微服务又包含了面向 PC 端的 Web 应用和面向移动端的 Wap 应用等方面设计。所以,使用这种组建方法,再根据不同功能进行模块划分,一个微服务项目工程大约包含如表 5-1 所示的模块结构。

表 5-1 微服务工程模块结构列表

序 号	模块名称	功能说明	类 型	备 注
1	*-object	查询对象设计	程序包	*表示工程名称
2	*-domain	业务领域设计	程序包	实体及持久化

续表

序 号	模块名称	功能说明	类 型	备 注
3	*-restapi	Restful 接口	微服务应用	
4	*-client	接口客户端封装	程序包	由 UI 应用使用
5	*-web	PC 端 Web UI	微服务应用	
6	*-wap	移动端 Web UI	微服务应用	

其中，各个模块之间的调用关系如图 5-1 所示。

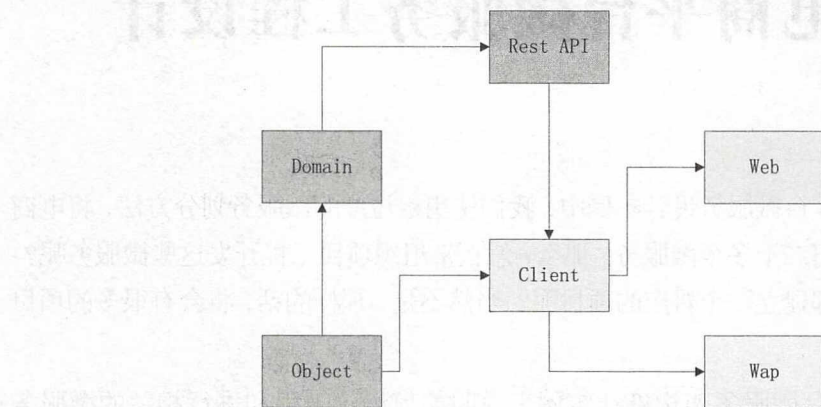


图 5-1 一个项目工程中各个模块的调用关系图

在这个调用关系中，Object 模块为业务领域设计和接口客户端设计提供查询对象定义，Domain 模块为 Rest API 应用提供业务领域设计的服务调用，Client 模块封装了对 Rest API 的调用，给 Web 和 Wap 应用提供高并发的接口调用方法。从各个模块的调用关系中可以看出，使用我们组建的工程结构将是各种模块的最优化组合方式。

使用上面的工程结构，就可以针对电商平台的各种业务类型来组建工程。例如，在类目服务中，我们可以组建一个类目微服务工程，它的模块结构如表 5-2 所示。

表 5-2 类目微服务工程模块结构列表

序 号	模块名称	功能说明	类 型	备 注
1	catalog-object	查询对象设计	程序包	
2	catalog-domain	业务领域设计	程序包	实体及持久化
3	catalog-restapi	Restful 接口服务	微服务应用	
4	catalog-client	接口客户端封装	程序包	由 UI 应用使用
5	catalog-web	PC 端 Web UI	微服务应用	
6	catalog-wap	移动端 Web UI	微服务应用	

5.2 电商平台微服务工程组建

使用上一节的工程模块结构设计，我们就可以按第3章的电商平台微服务设计实例，组建出如表5-3所示的电商平台的各个微服务工程。

表 5-3 电商平台微服务工程列表

序 号	项目名称	功能说明	使用数据库
1	catalog-microservice	类目微服务工程	catalogdb
2	goods-microservice	商品微服务工程	goodsdb
3	order-microservice	订单微服务工程	orderdb
4	shopingcart-microservice	购物车微服务工程	shopingcart
5	payment-microservice	支付微服务工程	paymentdb
6	logistics-microservice	物流微服务工程	logisticsdb
7	comment-microservice	评价微服务工程	commentdb
8	customer-microservice	顾客微服务工程	customerdb
9	member-microservice	会员微服务工程	memberdb
10	track-microservice	浏览记录微服务工程	trackdb
11	merchant-microservice	商家微服务工程	merchantdb
12	manage-microservice	平台管理微服务工程	managedb

经过工程的组建，电商平台的二十多个微服务就可以分布在十多个工程之中。其中，除了最后面两个微服务工程，即商家微服务工程和平台管理微服务工程的模块结构有些特别之外，其他大部分工程的模块结构基本上都相同。

5.3 数据库选型

每个微服务工程都可以具有各自独立的数据库，因此，每个工程都可以根据其自身的业务特殊性，选择合适的数据库。

其中，在浏览记录微服务工程中，是对用户浏览商品的足迹进行记录，它的数据量将会比较大，所以可以考虑使用 NoSQL 数据库，比如使用 MongoDB 会比较合适。而其他工程的数据库，基本上都可以使用 MySQL 数据库。

对于数据库的使用，在生产环境的安装和部署中，我们还将进行高可用和高性能的数据库集群设计。例如对于 MySQL 来说，通过使用主从设计、读写分离设计等方法，可以



构建成一个能够持续扩容的数据库集群架构。有关这方面的实现细节将在后续的部署篇的相关章节中介绍。不管数据库如何设计，它对于工程的调用来说是完全透明的，所以我们在工程中进行开发时，并不用花心思去理会数据库管理系统中复杂的设计。

5.4 微服务工程创建步骤

限于篇幅，本书并不能提供整个电商平台所有微服务工程的开发实例，但为了便于说明和演示，将会提供如表 5-4 所示的几个微服务工程的实例。这些实例工程包括了电商平台中的类目管理、商品管理、订单管理、商家管理和平台管理等业务功能，已经涵盖了本书所提倡的微服务架构设计方法及其一些先进技术的使用。通过这些实例的演练，读者可以熟练地使用微服务架构的开发方法。对于其他一些项目工程，可以留给读者作为练习使用。通过本书的学习，相信读者能够很容易地完成其他一些工程的创建和开发。

表 5-4 本书实例工程列表

序 号	工程名称	说 明	代 码 库
1	catalog-microservice	类目微服务工程	https://gitee.com/chenshaojian/catalog-microservice.git
2	goods-microservice	商品微服务工程	https://gitee.com/chenshaojian/goods-microservice.git
3	order-microservice	订单微服务工程	https://gitee.com/chenshaojian/order-microservice.git
4	merchant-microservice	商家微服务工程	https://gitee.com/chenshaojian/merchant-microservice.git
5	manage-microservice	平台管理微服务工程	https://gitee.com/chenshaojian/manage-microservice.git

如表 5-4 所示，本书的实例工程都存放在“开源中国”的码云代码库之中，可以通过 IDEA 检出或通过浏览器下载使用。

那么，这些工程是如何创建的呢？下面，我们介绍一下创建一个微服务工程的步骤，这些步骤将适用于上面每一个工程的创建过程。

首先，在 IDEA 的文件菜单中选择新建项目，这将打开一个新项目的对话框。然后在对话框的左面侧边栏中选择“Maven”，在顶端的“Project SDK”中选择已经安装的 JDK1.8，如图 5-2 所示。



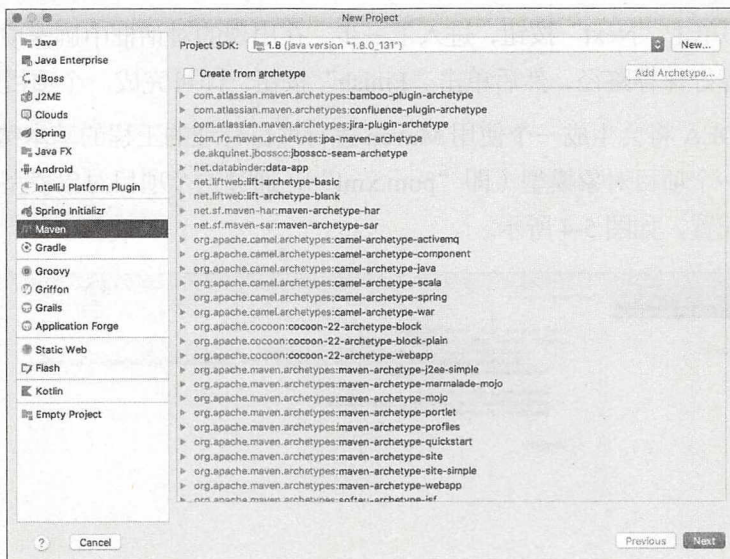


图 5-2 使用 Maven 创建微服务工程

在图 5-2 中单击“Next”按钮，进入下一步，在出现的对话框的“GroupId”中填写项目组织即程序包结构为“com.demo”，在“ArtifactId”中填写项目名称（例如“merchant-microservice”），如图 5-3 所示。

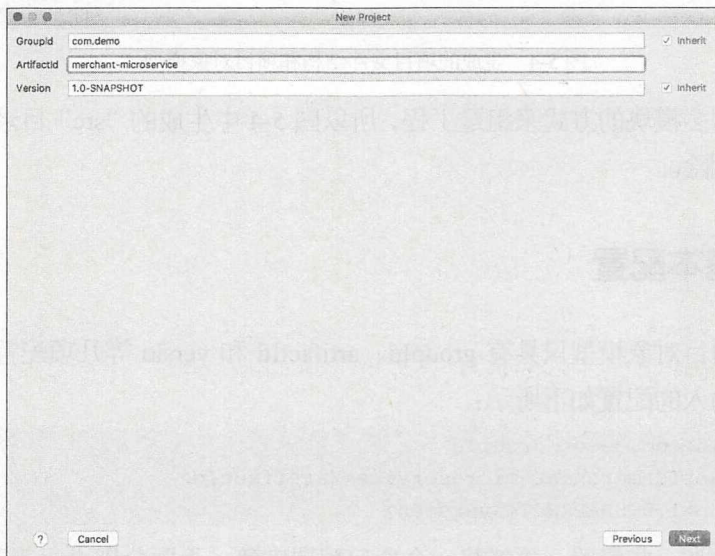


图 5-3 填写项目工程组织和项目名称

在图 5-3 中单击“Next”按钮，进入下一步，在出现的对话框中确保项目工程名称的正确性，并选择好保存路径，然后单击“Finish”按钮，即可完成一个项目的创建。

接下来，IDEA 将会生成一个使用 Maven 进行项目管理的工程的基本结构，在工程的根目录中包含一个项目对象模型（即“pom.xml”），通过这个项目对象模型，可以进行一些组件的引用配置，如图 5-4 所示。

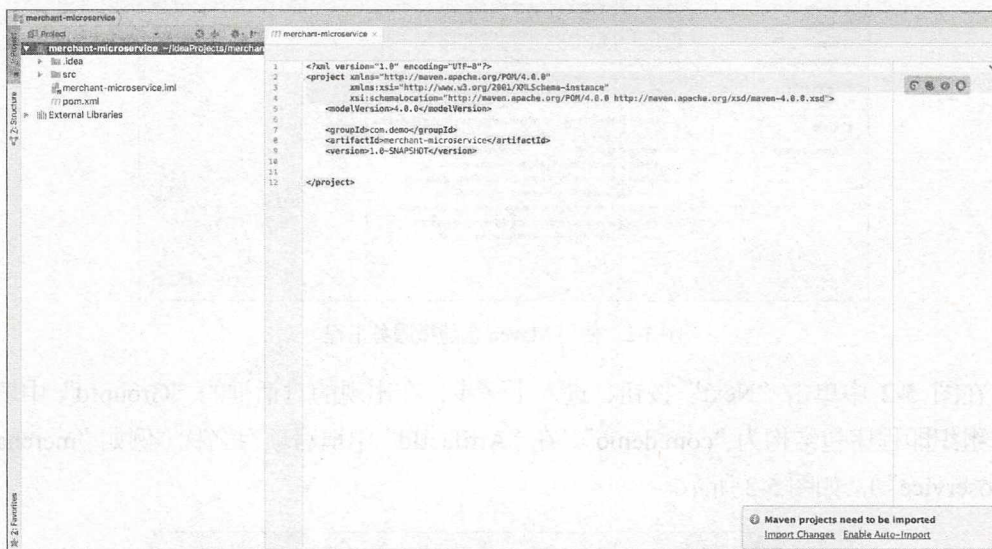


图 5-4 生成的项目文件结构和项目对象模型

我们将使用多模块的方式来组建工程，所以图 5-4 中生成的“src”目录将没有什么用处，可以将其删除。

5.5 项目基本配置

刚生成的项目对象模型只具有 groupId、artifactId 和 version 等几项配置，我们在创建项目的步骤中输入的配置如下所示：

```
<groupId>com.demo</groupId>
<artifactId>merchant-microservice</artifactId>
<version>1.0-SNAPSHOT</version>
```

现在还要增加一些配置，才能将一个项目配置完整。下面分别进行简要说明。

（1）使用属性来设置编译工具的版本和项目的编码方式，如下所示：


```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.
outputEncoding>
  <java.version>1.8</java.version>
</properties>

```

其中，源代码和输出编码都使用 UTF-8 字符集，这将保证在中文环境中源代码编辑和程序运行的日志输出中都不会出现乱码的情况。

(2) 使用父级配置，设定 Spring Boot 和 Spring Cloud 的版本号，这样，在后面创建的模块就将继承这个配置，并在引用相关组件时也可以不用再指定版本号，如下所示：

```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

其中，Spring Boot 的版本号为 1.5.9.RELEASE，Spring Cloud 的版本号为 Edgware.RELEASE。

(3) 使用下列一些通用配置，可以为所有模块设置默认的依赖引用，这些引用包含了 Spring Boot 开发框架的一些基本组件，包括自动配置和单元测试等。

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>

```



```

</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>

```

其中，将测试组件的范围指定为 `test`，将可保证这些组件的引用只是作为测试时使用，而不会被放进发行包之中。

(4) 使用统一的构建插件配置，如果模块中没有相关的构建配置，将默认使用这个配置。

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.20</version>
      <configuration>
        <skipTests>true</skipTests>
      </configuration>
    </plugin>
  </plugins>
</build>

```

其中测试插件 `surefire` 的配置忽略了在工程构建时对测试用例的调用。

将上面的各项配置综合起来，一个微服务项目的完整的配置如下所示：

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

```




```

<groupId>com.demo</groupId>
<artifactId>manage-microservice</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.
outputEncoding>
  <java.version>1.8</java.version>
</properties>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

```




```

</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.20</version>
      <configuration>
        <skipTests>true</skipTests>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>

```

完成项目对象模型即“pom.xml”文件的编辑之后，单击其右下角提示框中的“Import Changes”，将导入和下载配置中的相关组件，这将需要一定的时间，主要视网络情况而定。

5.6 创建模块

以上述创建的微服务工程“merchant-microservice”为例来创建模块。在 IDEA 中，将鼠标移至项目名称“merchant-microservice”的位置，单击鼠标右键，在弹出的菜单中选择“New Module”，在“New Module”对话框的“ArtifactId”中填写模块名称“merchant-object”，如图 5-5 所示。

在图 5-5 中单击“Next”按钮，进入下一步，在出现的对话框中确保模块名称和程序存放路径的正确性，然后单击“Finish”按钮，即可完成一个模块的创建步骤。

使用相同方法创建“merchant-domain”、“merchant-restapi”、“merchant-client”、“merchant-web”等模块，完成后的效果如图 5-6 所示。



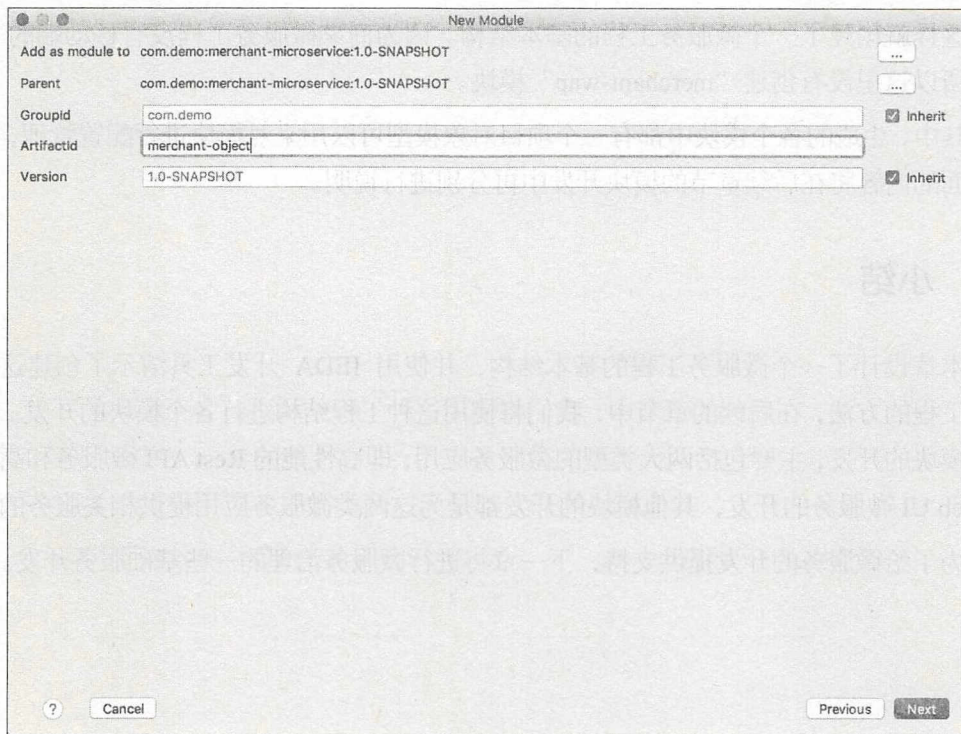


图 5-5 创建模块



图 5-6 创建了模块的微服务工程实例

这样就创建了一个微服务工程的基本结构。因为商家微服务工程没有移动端的 Web UI，所以这里没有创建“merchant-wap”模块。

其中，生成的各个模块中都有一个项目对象模型可以用来对模块进行配置管理，有关这方面的内容将在后续章节的模块开发中再分别进行说明。

5.7 小结

本章设计了一个微服务工程的基本结构，并使用 IEDA 开发工具演示了创建这种微服务工程的方法，在后续的章节中，我们将使用这种工程结构进行各个模块的开发。对于这些模块的开发，主要包括两大类型的微服务应用，即高性能的 Rest API 微服务和高并发的 Web UI 微服务的开发，其他模块的开发都是为这两类微服务应用提供相关服务的。

为了给微服务的开发提供支持，下一章将进行微服务治理的一些基础服务开发。



6

微服务治理基础服务开发

微服务将运行于云端或分布式环境之中，在这个环境中，如何有效地管理微服务，并维持微服务的正常通信，是通过微服务治理来实现的。有关微服务治理的内容包括如下几个方面的功能和服务：

- 服务配置管理。
- 服务注册管理。
- 服务路由管理。
- 服务调度管理。
- 服务监控管理。
- 服务跟踪管理。

本章将使用 Spring Cloud 工具套件来开发实施微服务治理的几个基础服务，这些服务本身也是一个微服务应用。这些服务包括配置管理中心、注册管理中心、微服务监控中心、聚合服务监控中心、服务跟踪分析中心等。其中，注册管理中心是微服务治理的核心，其他应用和组件将围绕注册管理中心来提供相关的服务。

这些基础服务的开发将通过一个项目工程进行管理，为此，创建一个基础服务工程，名字设定为“base-microserver”，并在这个工程中为各个服务建立一个相应的模块，各个模块的名称和功能如表 6-1 所示。



表 6-1 基础服务工程模块结构表

序 号	模 块	功 能
1	base-config	配置管理中心
2	base-eureka	注册管理中心
3	base-hystrix	微服务监控中心
4	base-turbina	聚合服务监控中心
5	base-zipkin	服务跟踪分析中心

其中,有关工程的创建、项目管理配置,以及各个模块的创建方法,与第 5 章的微服务工程创建的方法基本相同,这里不再赘述。下面,我们开始各个模块的开发。

这个基础服务工程的完整代码存放在“开源中国”的码云代码仓库之中,下面是链接地址:

```
https://gitee.com/chenshaojian/base-microservice.git
```

读者可以通过 Git 客户端检出或直接下载使用。

6.1 注册管理中心

注册管理中心是微服务治理的核心,其他治理服务和组件将以注册管理中心为基础提供相关的服务。Spring Cloud 工具套件中提供了可以使用 Zookeeper、Consul 和 Eureka 等多种选择的工具组件来创建和使用注册管理中心。这里从易于使用和性能方面考虑,将使用 Eureka 来创建注册管理中心。

6.1.1 创建注册管理中心

在项目“base-microserver”中使用模块“base-eureka”来创建注册管理中心。

使用 Eureka 组件创建一个注册管理中心是非常简单的,使用如下几个步骤就可以创建一个注册管理中心。

首先,在模块的项目配置 pom.xml 中引入“eureka-server”的依赖,如下所示:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
</dependencies>
```


然后，创建一个启动程序，名字设定为“EurekaApplication”，实现的代码如下所示：

```
package com.demo.base.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

在这个启动程序中，我们使用注解“@EnableEurekaServer”将这个应用设置为注册服务器。

最后，在工程的配置文件 application.yml 中增加如下所示的配置：

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:${server.port}/eureka/

spring.cloud.config.discovery.enabled: true
```

这样就完成了注册管理中心的创建。

这个配置将注册管理中心的端口设置为“8761”。后面以 eureka 开头的配置都是注册管理中心的客户端配置，所以在这里将 registerWithEureka 和 fetchRegistry 都设置为 false，表示本应用实例将不在注册管理中心中进行注册。

其中，registerWithEureka 表示是否将本实例在 Eureka Server 中进行注册，默认为 true，fetchRegistry 表示是否从 Eureka Server 中取得本实例的注册信息，默认为 true。如果将注册管理中心进行多实例发布，可以将 registerWithEureka 设置为 true。另外，defaultZone 表示从默认分区中访问 Eureka Server。



6.1.2 运行注册管理中心

注册管理中心使用内存来管理实例元数据，所以不用使用数据库。启动 EurekaApplication 程序即可运行注册管理中心。启动成功之后，通过浏览器打开如下链接，即可查看注册管理中心的运行情况：

<http://localhost:8761/>

打开链接后如图 6-1 所示，因为这时还没有客户端注册实例，所以实例列表为空。

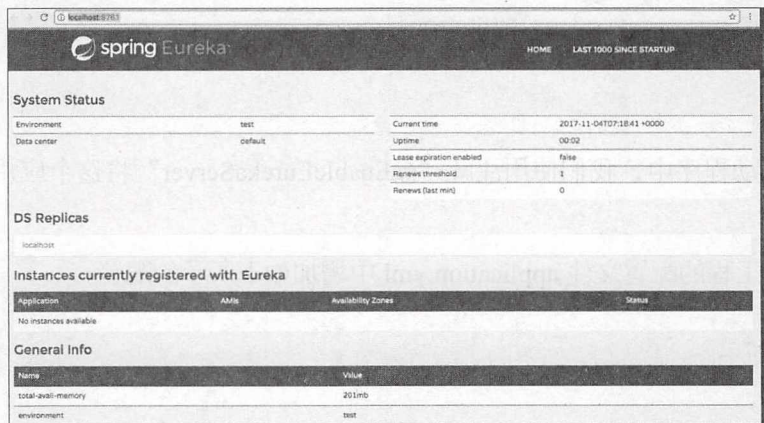


图 6-1 注册管理中心运行效果

6.1.3 微服务怎样使用注册管理中心

每个微服务是怎样来使用注册管理中心的呢？主要通过以下三个步骤来实现。

首先，在项目中增加 Eureka 的相关依赖引用，如下所示：

```
<dependencies>
  <!--注册服务-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <!--路由服务-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
  </dependency>
</dependencies>
```

这里不但引用了 Eureka 组件，也引用了 Zuul 组件，Zuul 组件提供智能路由的服务。引用 Zuul 组件之后，Ribbon 组件也会被自动引用进来，它提供了负载均衡的功能。

然后，在微服务的主程序中增加如下所示的三个注解：

```
@EnableDiscoveryClient
@EnableZuulProxy
@EnableHystrix
```

其中，第一个注解表示使用客户端服务发现的功能；第二个注解启用 ZuulProxy 代理以使用智能路由的功能；第三个注解使用 Hystrix 的断路器功能以提供服务降级和容错的机制。

最后，在工程的相关模块配置中增加如下所示的配置，即可接入注册管理中心：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    instance:
      preferIpAddress: true
  spring:
    application:
      name: catalogapi
```

其中，preferIpAddress 的配置使用了客户端的 IP 地址来注册，这里也可以使用 hostname 来设置主机名或 IP。最后的“catalogapi”为实例的名称，这个名称必须保证唯一性。

完成上面的设置，并确保注册管理中心已经启动，然后启动微服务，就可以在注册管理中心之中看到已经注册的实例，如图 6-2 所示。

The screenshot shows the Eureka web interface in a browser. The title is "Instances currently registered with Eureka". Below the title is a table with columns: Application, AMIs, Availability Zones, and Status. The first row shows "CATALOGAPI", "n/a (1)", "(1)", and "UP (1) - 192.168.0.104:catalogapi:9091". Below this table is a "General Info" section with a table of system metrics. At the bottom is an "Instance Info" section with a table showing IP address and status.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CATALOGAPI	n/a (1)	(1)	UP (1) - 192.168.0.104:catalogapi:9091

General Info	
Name	Value
total-avail-memory	152mb
environment	test
num-of-cpus	2
current-memory-usage	74mb (48%)
server-up-time	1 day 16:16
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/
available-replicas	

Instance Info	
Name	Value
ipAddr	172.17.0.2
status	UP

图 6-2 已经有实例注册的注册管理中心

图 6-2 中的服务实例“catalogapi”是电商平台中类目管理的 RestAPI 微服务。

在后面的开发调试中，都将用到注册管理中心，所以在没有特别说明的情况下，我们都要启动一个注册管理中心以备使用。

6.1.4 构建高可用的注册管理中心

从上面的介绍中可知，注册管理中心在微服务治理中的地位是如何至关重要，所以要保证注册管理中心的稳定性和可靠性，避免出现单点故障的问题，单靠一个注册管理中心是不够的。

将注册管理中心进行多副本发布，让不同的副本之间进行互相注册，就可以构建一个高可用的注册管理中心集群。注册管理中心这种集群架构还有一个很特别的地方，即在集群中并不存在主从关系，注册管理中心的每个副本之间的关系都是平等关系。在注册管理中心集群中，不同的副本之间会定期将已经注册的应用实例进行互相复制，从而确保每个副本都具有一份完整的应用注册实例列表。

通过更改注册管理中心的配置，然后分别打包发布，就可以构建一个高可用的注册管理中心集群。

例如，我们使用如下所示的 3 个 IP 来构建一个注册管理中心集群：

```
192.168.1.31
192.168.1.32
192.168.1.33
```

那么只要使用上面创建的注册管理中心，通过更改其相关的配置就可以实现。

例如，第一个注册管理中心可以使用如下所示的配置：

```
server:
  port: 8761

eureka:
  instance:
    hostname: 192.168.1.31
  client:
    registerWithEureka: true
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://192.168.1.32:${server.port}/eureka/,http://192.168.1.33:${server.port}/eureka/
```




```
spring.cloud.config.discovery.enabled: true
```

其中，将“registerWithEureka”设置为“true”，然后，同时在其他两个注册管理中心中进行注册。

另外，注册管理中心的名称也必须跟着进行更改，如下所示：

```
spring:
  application:
    name: eurekal
```

而其他两个注册管理中心，就可以参照上面的配置进行设置。比如它们的名字可以分别设置为“eureka2”和“eureka3”。

在“base-eureka”模块实例中，我们是通过使用 Profiles 的方式来管理不同副本的配置的，这样，选择不同的 Profiles 进行打包，就可以实现上面的设计。

如果服务器分布在不同的地区或不同机房之中，注册管理中心还可以进行分区设计。通过分区设计，不但可以提高微服务的高可用性，而且也可以提高服务之间的调用效率。如果在不同分区之中存在相同的服务，服务之间的调用将会根据最接近的分区进行选择。

我们这里只用到一个默认区域即“defaultZone”。

6.2 配置管理中心

配置管理中心可以为所有微服务提供一个统一的配置管理服务。微服务可以使用本地工程的配置，也可以使用配置管理中心的配置，当这两方面具有相同的配置项时，系统默认优先使用配置管理中心所提供的配置。

6.2.1 创建配置管理中心

在基础服务工程“base-microservice”中创建一个模块，将模块名称设置为“base-config”，用来创建配置管理中心。创建配置管理中心的步骤如下。

首先，在模块的项目管理中增加如下所示的依赖引用：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
  </dependency>
```



```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
</dependencies>

```

这个依赖引用的各个组件的功能说明如下：

- “config-server” 是一个配置管理服务器，可用来创建配置管理中心。
- “eureka” 是一个服务注册与发现的工具组件，提供了注册管理中心客户端的功能。
- “bus-amqp” 是一个消息总路线服务组件，可为配置管理中心提供使用消息进行通信的功能。

其次，创建一个启动程序，实现代码如下所示：

```

@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class ConfigApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }
}

```

这里使用了几个注解来设置这个应用程序，其中注解 “@EnableConfigServer” 将这个应用设置为配置管理服务器，注解 “@EnableDiscoveryClient” 将这个应用设置为注册管理中心的客户端，可以使用服务发现的功能。

然后，在工程的配置文件 application.yml 中增加如下所示的配置：

```

server:
  port: 8888

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

management.security.enabled: false

```



在这个配置中，实现如下几项设定：

- 将应用的端口设置为“8888”。
- 连接了本地的注册管理中心。
- 关闭安全设置。因为配置中心将在内网中使用，也不对外提供服务，所以为了方便调用，这里将安全管理功能设置为“false”。

最后，再在工程中增加一个名字为 `bootstrap.yml` 的配置文件，并使用如下所示的配置：

```
spring:
  application:
    name: config

  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/chenshaojian/base-config-repo.git
          username:
          password:

  rabbitmq:
    addresses: amqp://localhost:5672
    username: alan
    password: alan
```

在上面的这个配置中，我们做了如下几项设定：

(1) 使用 Git 仓库来存放客户端的配置文件，它的链接地址设置如下：

```
https://gitee.com/chenshaojian/base-config-repo.git。
```

为了方便测试，已经公开了这个代码仓库的权限。如果是实际使用可以将代码仓库设为私有，并设定具有访问权限的用户和密码。

(2) 设定 RabbitMQ 服务器的配置。其中 RabbitMQ 服务器的地址、用户和密码根据实际安装的情况进行设定。

有关 RabbitMQ 服务器的安装和用户的管理可参考第 14 章中的相关说明。

在应用启动时，配置文件“`bootstrap.yml`”将在“`application.yml`”文件之前被加载。在使用配置管理中心的客户端中，都有一个“`bootstrap.yml`”文件，用来配置应用的名称和连接配置管理中心的设置。



如果已经启动注册管理中心，现在可以启动配置管理中心进行测试。

配置管理中心并没有提供管理操作界面，可以通过观察控制台查看启动状态。

配置管理中心将使用 RabbitMQ 与客户端进行通信，如果不能正常连接 RabbitMQ 服务器，程序将会不停地发出类似于如下所示的警告：

```
Consumer raised exception, processing can restart if the connection factory
supports it.
.....
```

6.2.2 微服务如何使用配置管理中心

微服务要使用配置管理中心提供的功能，必须先连接到注册管理中心，然后使用下列的步骤进行设置。

首先，在应用的项目管理配置 `pom.xml` 文件中增加如下所示的依赖引用：

```
<dependencies>
    .....
    <!--配置服务-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>

    <!-- 消息服务-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bus-amqp</artifactId>
    </dependency>
    .....
</dependencies>
```

其中，`config` 组件提供了使用配置管理中心的功能，`bus-amqp` 提供了使用消息通道的功能。

然后，在工程中增加一个名称为 `bootstrap.yml` 的配置文件，并使用类似如下所示的设置：

```
spring:
  application:
    name: turbine
```



```
cloud:
  config:
    uri: http://localhost:8888

rabbitmq:
  addresses: amqp://localhost:5672
  username: alan
  password: alan
```

这个配置做了如下几项设定：

- 将应用的名称设置为“turbine”。
- 连接配置管理中心。
- 设定 RabbitMQ 服务器的各项参数。

这样，当在配置管理中心的文件仓库中具有“application.yml”和“turbine.yml”文件时，其中的配置信息就将会被这个应用使用。需要注意的是，“application.yml”文件将会被所有使用配置管理中心的应用使用，而以应用名称为开头的配置文件只为应用所独有。这里，配置文件的扩展名也可以是“.properties”。

另外，如果我们在上面的配置中使用“profiles”这个参数，还可以在文件名中使用后缀来设定文件仓库中的文件。例如使用如下所示的配置：

```
spring:
  application:
    name: turbine
  profiles:
    active: develop
```

那么这个应用独有的配置文件应该为“turbine-develop.yml”或“turbine-develop.properties”。

所以，在配置应用名称时，最好不要包含“-”这个连接符，以免引起未知的错误。

通过上面的设置之后，我们以前在应用本地的配置文件“application.yml”中所有的配置，都可以由配置管理中心进行统一管理。

使用配置管理中心之后，可以在应用不重启的情况下，在线更新应用的配置信息。

6.2.3 在线更新配置信息

由于配置管理中心没有提供管理操作界面，所以实现在线更新可以通过使用指令来操作。



如果要更新所有客户端的配置，可以使用如下指令：

```
curl -X POST http://localhost:8888/bus/refresh
```

这条指令可以在运行配置管理中心的机器上执行。如果要远程执行，可以将“localhost”改为相关的 IP 地址。

如果要更新某一个应用的配置，可以使用如下的指令执行：

```
curl -X POST http://localhost:8888/bus/refresh?destination=orderweb:**
```

其中“orderweb”表示需要进行在线更新的目标应用。

更新一个应用的配置，也可以使用类似如下所示的指令，在运行应用的机器中执行：

```
curl -X POST http://localhost:9001/refresh
```

需要注意的是，对于一个应用来说，通过在线更新管理，并不是所有的配置信息都能生效的。有些配置，例如有关连接数据源的配置，在应用启动时就已经建立了数据库的连接，而在线更新配置并不能更改这种配置。

还有就是如果应用的配置是通过程序来动态读取的，那么要使之能够取得在线更新的配置信息，必须在程序的开头加上这个“@RefreshScope”注解才能生效。例如：

```
@RestController
@RefreshScope
public class TestController {
    @Value("${cloud.sample.msg}") String msg;
    .....
}
```

这样，配置项“cloud.sample.msg”的值就可以通过配置管理中心进行在线更新了。

在线更新功能在某些方面的使用是非常方便的。例如，下面将要介绍的聚合服务监控中心，就可以通过在线更新功能，根据需要随时更改需要监控的服务。

6.3 微服务监控中心

在众多正在运行的微服务中，我们必须做到随时掌握每一个服务的运行情况及其健康状况，才能保证整个平台的稳定性和可靠性。使用 Hystrix 断路器仪表盘功能就可以创建一个监控中心，实现在线监控微服务的运行状态。



6.3.1 使用断路器仪表盘实现监控

在基础服务工程“base-microservice”中创建一个模块，名称设为“base-hystrix”，用来创建一个监控中心。

首先，在项目的配置管理中增加如下所示的依赖配置，即引用 Hystrix 组件的断路器仪表盘“hystrix-dashboard”。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
  </dependency>
</dependencies>
```

接着创建一个启动程序，命名为“HystrixApplication”，如下所示：

```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class HystrixApplication {
    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, args);
    }
}
```

这里，使用了注解“@Controller”创建了一个控制器，设定对主页的链接转发到“/hystrix”之中。使用注解“@EnableHystrixDashboard”启用断路器仪表盘的功能。

然后，在应用的配置文件“application.yml”中，增加如下所示的配置项：

```
server:
  port: 7979

endpoints:
  restart:
    enabled: true
  shutdown:
    enabled: true
```

即主要配置了应用的端口为“7979”。



现在启动这个应用，使用如下所示的链接，就能打开监控中心的控制台：

<http://localhost:7979>

图 6-3 所示为监控中心控制台的首页。

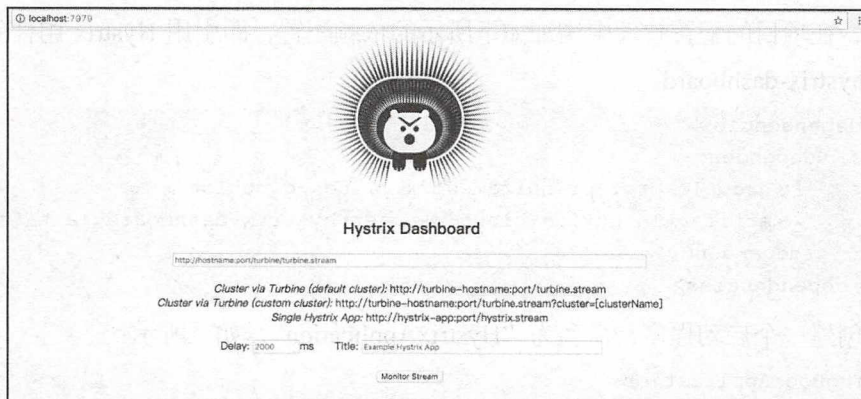


图 6-3 微服务监控中心控制台首页

在图 6-3 的输入框中，按其格式输入需要监控的微服务，然后单击“Monitor Stream”按钮，即可打开一个微服务的监控界面，这个监控界面通过断路器仪表盘来显示微服务的运行情况。

例如，我们输入如下的链接，就可以监控其正在运行的微服务：

<http://192.168.1.98/hystrix.stream>

图 6-4 所示为一个微服务运行正常时的断路器仪表盘状态。

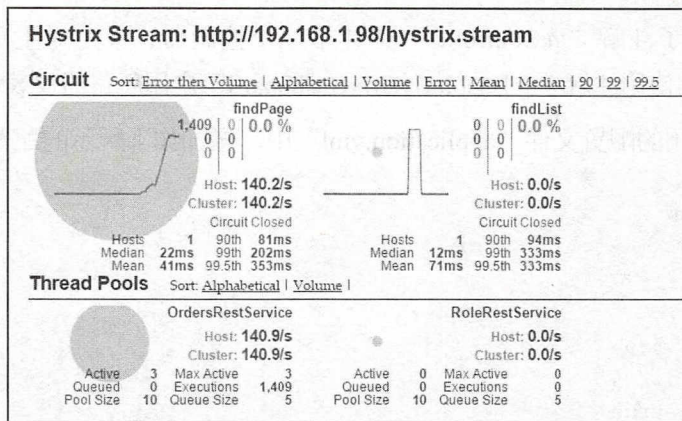


图 6-4 服务运行正常时的断路器仪表盘状态

这是一个正在进行性能测试的微服务的运行情况，图 6-4 中的圆形会随着并发量的变化而出现膨胀和缩小的状态，当服务运行正常时，断路器处于闭合状态之中。

当服务出现故障时，断路器就会被打开，圆形图形会变为红色，并在旁边显示服务的故障比率，如图 6-5 所示。

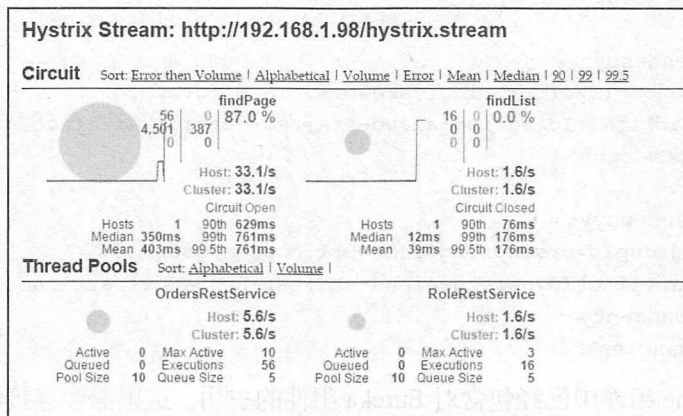


图 6-5 服务出现故障时的断路器仪表盘状态

6.3.2 聚合服务监控管理中心

上面的监控中心能够单独针对每一个微服务进行监控。当需要监控不同服务时，将要切换不同的界面，这在操作上会有点不方便。下面来创建一个聚合服务监控中心，可以在一个界面中同时监控多个微服务。

在基础工程“base-microservice”中新建一个模块，模块名字设定为“base-turbine”，用来创建一个聚合服务监控中心。

这个模块将使用 Spring Cloud Turbine 组件，它提供了聚合服务监控的功能。在模块中增加如下所示的依赖引用：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
```



```
<artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
</dependencies>
```

其中，Turbine 组件中已经包含对 Eureka 组件的引用。这里需要连接注册管理中心和配置管理中心，并使用配置管理中心来设定需要监控的微服务。这样就可以动态更改需要监控的微服务列表。

创建一个启动程序，名字设定为“TurbineApplication”，如下所示：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableTurbine
@EnableHystrixDashboard
@Controller
public class TurbineApplication {
    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    public static void main(String[] args) {
        SpringApplication.run(TurbineApplication.class, args);
    }
}
```

这个程序与上节的监控中心的启动程序有点类似，只是多了两个注解，一个使用了 Eureka 的服务发现功能，另一个使用了 Turbine 聚合监控的功能。



在工程的配置文件“application.yml”中增加如下所示的配置：

```
server:
  port: 8989

management:
  port: 8990

eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
    non-secure-port: ${PORT:8989}
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

turbine:
  appConfig: orderapi,orderweb,orderwap
  aggregator:
    clusterConfig: default
  clusterNameExpression: new String("default")
```

其中，我们设定服务的端口为“8989”，并连接了注册管理中心，在“turbine”的设置中，通过“appConfig”来指定需要监控的服务。这里设定了需要监控的服务为：

- orderapi。
- orderweb。
- orderwap。

增加一个“bootstrap.yml”配置文件，并设置如下所示的内容以连接配置管理中心：

```
spring:
  application:
    name: turbine

  cloud:
    config:
      uri: http://localhost:8888

  rabbitmq:
    addresses: amqp://localhost:5672
    username: alan
    password: alan
```



现在启动应用进行测试。

启动成功后，输入如下的链接地址，就可以打开控制台：

```
http://localhost:8989/
```

你会发现，这个控制台与上节的控制台是一个完全一样的页面。不过，这个监控中心的功能是有些不一样的，输入如下网址即可打开聚合服务的监控界面：

```
http://localhost:8989/turbine.stream
```

如果我们所监控的服务都已经启动并且正在运行之中，你将可以看到类似于如图 6-6 所示的多服务监控的界面。

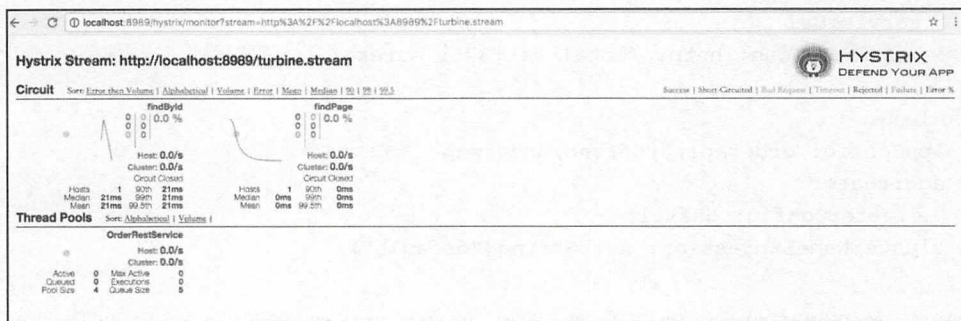


图 6-6 多服务监控运行效果

在应用的控制台中，也可以看到类似于如下的输出信息：

```
Fetching instance list for apps: [orderapi, orderweb, orderwap]
Fetching instances for app: orderapi
Received instance list for app: orderapi, size=1
Fetching instances for app: orderweb
Received instance list for app: orderweb, size=1
Fetching instances for app: orderwap
Eureka returned null for app: orderwap
Retrieved hosts from InstanceDiscovery: 2
Found hosts that have been previously terminated: 0
Hosts up:2, hosts down: 0
```

从这个输出信息中可知，我们所监控的三个服务之中，只有两个服务是正在运行的，分别是“orderapi”和“orderweb”，而“orderwap”却没有正常运行。这个时候，在注册管理中心中，也可以看到类似于如图 6-7 所示的已经注册的在线应用，可以看出“orderwap”并不在列表之中。

Application	AMs	Availability Zones	Status
CONFIG	n/a (1)	(1)	UP (1) - 192.168.0.101:config:8888
ORDERAPI	n/a (1)	(1)	UP (1) - 192.168.0.101:orderapi:9095
ORDERWEB	n/a (1)	(1)	UP (1) - 192.168.0.101:orderweb:8095
TURBINE	n/a (1)	(1)	UP (1) - 192.168.0.101:turbine:8989

图 6-7 注册管理中心服务列表

在任何时候，我们都可以通过更改配置管理中心的文件库中的“turbine.yml”这个配置文件，在线更新聚合服务监控中心所监控的服务列表。当然，更新配置文件之后，不要忘记执行如下指令，以使配置文件更新立即生效：

```
curl -X POST http://localhost:8888/bus/refresh
```

执行这个指令之后，我们也能在 Turbine 应用的控制台中看到如下的输出信息：

```
Located property source: CompositePropertySource [name='configService',
propertySources=[MapPropertySource [name='configClient'], MapPropertySource
[name='https://gitee.com/chenshaojian/base-config-repo.git/turbine.yml']]
```

这表明，监控中心已经导入了新的配置信息，并将按照新的服务列表进行监控。

所以，使用聚合服务监控中心可以对整个平台的微服务，根据需要实现实时监控。

图 6-8 所示为在一个测试环境之中聚合服务监控中心运行监控的实例。

Hystrix Stream: http://192.168.1.27:8989/turbine.stream											
Circuit Sort: Error then Volume Alphabetical Volume Error Mean Median 90 99 99.5											
findByNames				UserClient#findByName(String)				RoleClient#findList()			
0 0 0.0 %				0 0 0.0 %				0 0 0.0 %			
0 0				0 0				0 0			
Host: NaN.0/s				Host: NaN.0/s				Host: 0.0/s			
Cluster: NaN.0/s				Cluster: NaN.0/s				Cluster: 0.0/s			
Circuit Closed				Circuit Closed				Circuit Closed			
Hosts	1	90th	0ms	Hosts	1	90th	0ms	Hosts	2	90th	0ms
Median	0ms	99th	0ms	Median	0ms	99th	0ms	Median	0ms	99th	0ms
Mean	0ms	99.5th	0ms	Mean	0ms	99.5th	0ms	Mean	0ms	99.5th	0ms
Thread Pools Sort: Alphabetical Volume											
UserRestService				merchantapi				RoleRestService			
Host: NaN.0/s				Host: 0.0/s				Host: 0.0/s			
Cluster: NaN.0/s				Cluster: 0.0/s				Cluster: 0.0/s			
Active	0	Max Active	0	Active	0	Max Active	0	Active	0	Max Active	0
Queued	0	Queued	0	Queued	0	Queued	0	Queued	0	Queued	0
Pool Size	0	Queue Size	0	Pool Size	20	Queue Size	5	Pool Size	20	Queue Size	5

图 6-8 聚合服务监控测试实例

6.4 服务跟踪分析中心

除了可以对微服务运行情况进行监控之外,我们也可以对微服务的调用线路和运行轨迹进行跟踪和分析。使用 Spring Cloud 的 Sleuth 组件的日志收集功能,并结合 Zipkin Server 一起使用,就可以很方便地建立起一个服务跟踪分析中心。

6.4.1 创建服务跟踪分析中心

在基础服务工程“base-microservice”中创建一个模块,名字设定为“base-zipkin”,用来创建一个服务跟踪分析中心。

首先在模块的项目管理 pom.xml 中增加如下所示的依赖配置:

```
<dependencies>
  <dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
  </dependency>
  <dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>

  <dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
  </dependency>
  <dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-metrics-event-stream</artifactId>
  </dependency>
</dependencies>
```




```

        <groupId>com.netflix.hystrix</groupId>
        <artifactId>hystrix-javanica</artifactId>
    </dependency>

    <!--mysql-->
    <dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-autoconfigure-storage-mysql</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.2</version>
    </dependency>
</dependencies>

```

这个依赖配置引用了如下一些组件：

- Zipkin 的服务器和 UI 组件。
- Spring Cloud 工具套件中的 Config、Eureka 和 Hystrix 等几个组件。
- MySQL 数据库及其 Druid 数据源等组件。

接着，创建一个启动程序，命名为“ZipkinServerApplication”，编写如下代码：

```

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableZipkinServer
public class ZipkinServerApplication {
    public static void main(String[] args) throws Exception {
        new SpringApplicationBuilder(ZipkinServerApplication.class).web
(true).run(args);
    }
}

```

其中，在前面的注解中，加入注册管理中心客户端和使用断路器的功能，并且将这个



应用设定为一个 Zipkin 服务器。

然后，在工程的配置文件“application.yml”中增加如下所示的配置：

```
server:
  port: 9987

# datasource
spring:
  datasource:
    schema: classpath:/mysql_init.sql
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/myzipkin?characterEncoding=utf8&
    useSSL=false
    username: root
    password: 12345678
    initialize: true
    continueOnError: true
    # 初始化大小, 最小, 最大
    initialSize: 5
    minIdle: 5
    maxActive: 20
    # 配置获取连接等待超时的时间
    maxWait: 60000
    # 配置间隔多久才进行一次检测, 检测需要关闭的空闲连接, 单位是毫秒
    timeBetweenEvictionRunsMillis: 60000
    # 配置一个连接在池中最小生存的时间, 单位是毫秒
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    # 打开 PSCache, 并且指定每个连接上 PSCache 的大小
    poolPreparedStatements: true
    maxPoolPreparedStatementPerConnectionSize: 20
    # 配置监控统计拦截的 filters, 去掉后监控界面 SQL 将无法统计, 'wall' 用于防火墙
    filters: stat,wall,log4j
    # 通过 connectProperties 属性来打开 mergeSql 功能; 慢 SQL 记录
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000

#不对本身跟踪
sleuth.enabled: false
```



```
# zipkin config
zipkin:
  storage:
    type: mysql

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

#ribbon config
ribbon.ConnectTimeout: 5000
ribbon.ReadTimeout: 10000
```

这个配置做了如下几项设置：

- 设定服务跟踪分析中心的服务端口为“9987”。
- 使用 Druid 配置了数据源连接。
- 指定存储跟踪信息的数据库为“mysql”。
- 配置了连接注册管理中心的参数。
- 配置了连接所有服务的超时设定。

根据数据源的配置信息，存储跟踪信息的数据库设为“myzipkin”，在工程中还准备了一个创建数据库及其表结构的脚本，如果数据库和表结构不存在，将在工程启动时进行自动创建。所以这里只要保证具有一个访问本地数据库并且具有完全权限的用户就可以了。如果没有这个用户及其相关的权限，可以在本地数据库中使用如下脚本创建一个：

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY '12345678';
```

最后，再在工程中增加一个“bootstrap.yml”配置文件，用来连接配置管理中心，文件内容设置如下：

```
spring:
  application:
    name: zipkin
  cloud:
    config:
      uri: http://localhost:8888

  rabbitmq:
    addresses: amqp://localhost:5672
    username: alan
    password: alan
```




```
encrypt:
  failOnError: false
```

这样就完成了一个服务跟踪分析中心的创建工作。

现在启动应用程序，启动成功后在浏览器中输入如下的链接地址，即可打开服务跟踪分析中心的操作界面：

`http://localhost:9987。`

如图 6-9 所示为打开服务跟踪分析中心的首页界面。

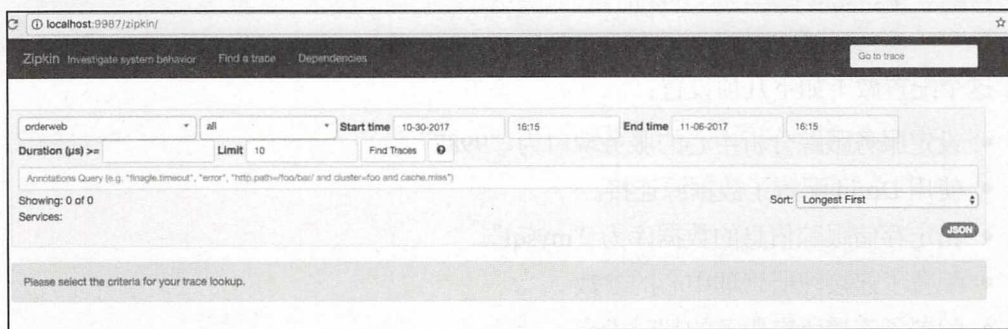


图 6-9 服务跟踪分析中心首页

因为现在还没有跟踪任何微服务，所以在服务跟踪分析中心之中还不能查询到任何信息。

6.4.2 在微服务中启用服务跟踪功能

一个微服务要使用服务跟踪分析中心的功能，只要引入跟踪服务的相关组件和进行一些简单配置就可以实现。通过如下步骤的设置，即可以在一个微服务应用之中启用服务跟踪功能。

首先，在微服务应用的项目管理之中增加如下所示的依赖引用：

```
<dependencies>
  .....
  <!-- 跟踪 -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
  </dependency>
```




```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
</dependencies>
```

即引用了组件“sleuth”和“zipkin”。

然后，在项目的配置文件“application.yml”中增加如下所示的配置项：

```
spring:
  zipkin:
    enabled: true
    base-url: http://localhost:9987
```

其中，“base-url”指定的是跟踪服务器的链接地址和端口。

这样，这个微服务就启用了服务跟踪的功能。

例如，我们将订单服务中的“orderapi”和“orderweb”（这些服务将在后续的章节中开发）加入服务跟踪的功能，并将应用启动起来。

现在我们在两个应用中做一下简单的数据查询等操作。然后，就可以在跟踪服务分析中心中看看服务跟踪的效果了。

在上节的服务跟踪管理中心的操作界面中单击“Find Traces”按钮，即可查出服务的跟踪信息，如图 6-10 所示。

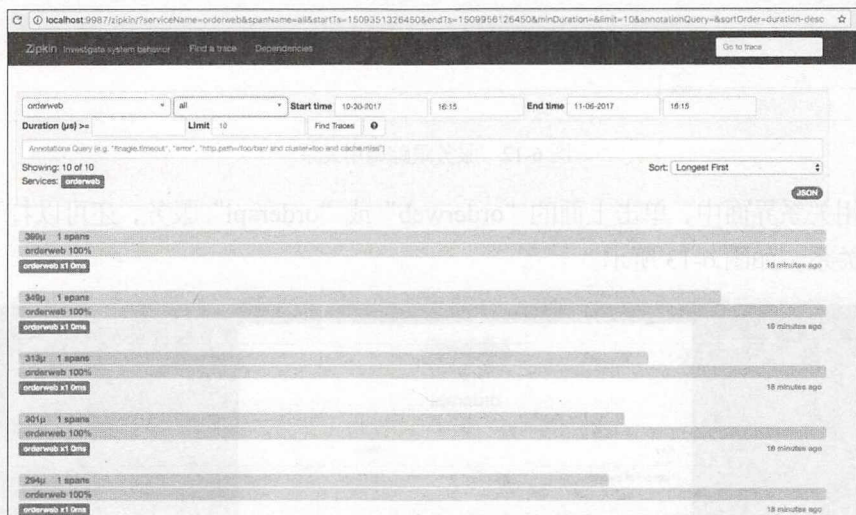


图 6-10 服务跟踪查询结果

在图 6-10 中单击右边的“JSON”按钮，可以打开如图 6-11 所示的记录。

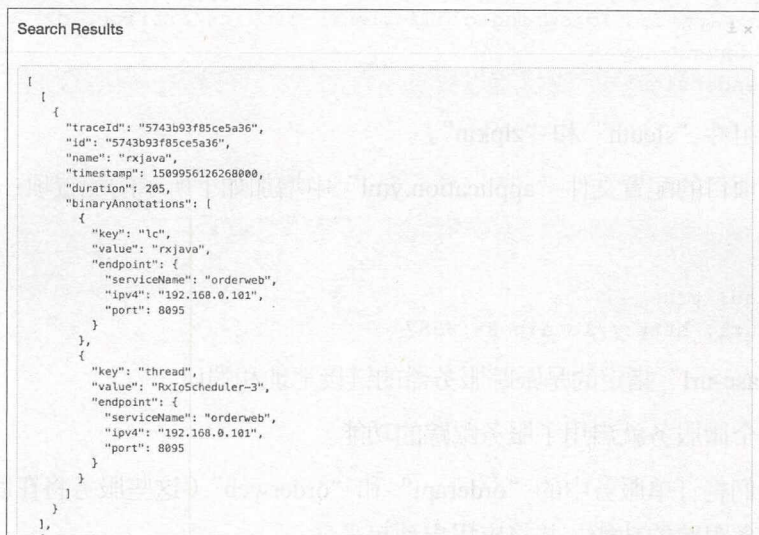


图 6-11 服务跟踪访问记录

单击分析中心首页上面的“Dependencies”，可以打开如图 6-12 所示的服务调用关系和线路，即服务“orderweb”调用了服务“orderapi”。

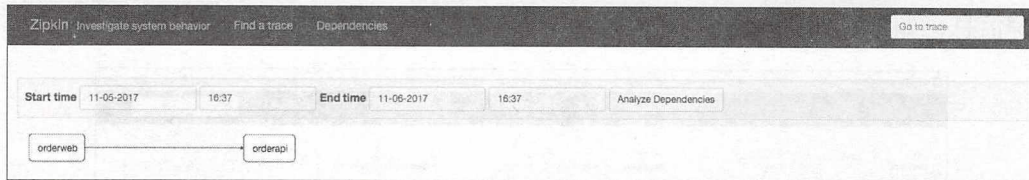


图 6-12 服务跟踪调用关系

在调用关系界面中，单击上面的“orderweb”或“orderapi”服务，还可以打开它们之间的调用关系，如图 6-13 所示。

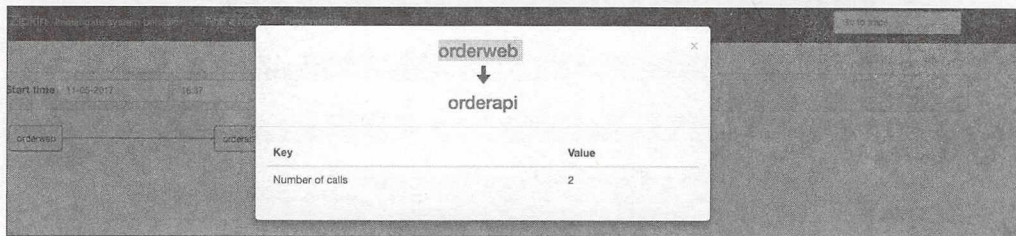


图 6-13 服务调用关系相关数据



所以，使用 Zipkin 跟踪服务，可以通过可视化的图形界面，非常清晰地查询微服务之间互相调用的线路及其详细情况，使用这些情况可以为故障分析提供强有力的帮助。

6.5 日志分析平台

除了对微服务的运行和相互调用可以使用监控和跟踪之外，微服务的输出日志也是故障分析中最直接的入口和切实依据。但是要在每个微服务的控制台上去看日志是很不方便的，特别是微服务不但将使用 Docker 来发布，并且分布在很多不同的服务器上，这更加使日志的查看更为不便。

所以，我们将使用一个日志分析平台，将所有微服务的日志收集起来，进行集中管理，然后提供统一的管理平台进行查询和分析。

6.5.1 创建日志分析平台

日志分析平台 ELK 分别由 Elasticsearch、Logstash 和 Kibana 三个服务所组成。其中，Elasticsearch 负责日志存储并提供搜索功能，Logstash 负责日志收集，Kibana 提供 Web 查询操作界面。这三个服务都是开源的，可以使用 Docker 进行安装。有关 ELK 安装说明请参照第 12 章的介绍。

6.5.2 使用日志分析平台

在微服务的工程中增加如下所示的依赖引用，即可以在应用中使用日志分析平台提供的日志收集功能。

```
<dependencies>
.....
<!--日志服务-->
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.10</version>
</dependency>
</dependencies>
```

然后，在应用中增加一个“logback.xml”配置文件，内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
```




```

<configuration>
  <appender name="stash" class="net.logstash.logback.appender.LogstashTcp
SocketAppender">
    <destination>192.168.1.28:5000</destination>
    <encoder charset="UTF-8" class="net.logstash.logback.encoder.
LogstashEncoder" />
  </appender>

  <appender name="async" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="stash" />
  </appender>

  <root level="info">
    <appender-ref ref="stash" />
  </root>
</configuration>

```

这个配置假设日志收集服务器的 IP 地址为“192.168.1.28”。

这样，应用启动之后，就可以通过如下链接打开日志分析平台：

<http://192.168.1.28:5601>

在日志分析平台中，可以查询每个应用的日志输出，如图 6-14 所示。

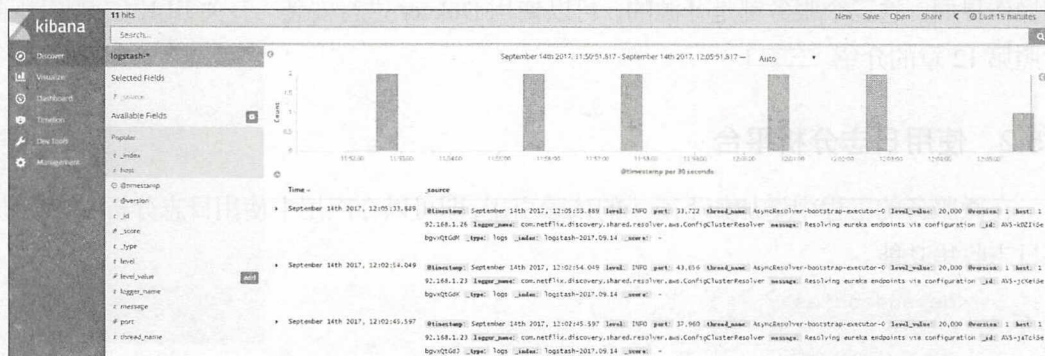


图 6-14 Kibana 日志查询界面

6.6 小结

本章通过一个基础服务工程，创建了注册管理中心、配置管理中心、服务监控管理中心、服务跟踪分析中心等微服务治理的基础服务，并为各个服务的使用进行了详细的说明。

在这个工程的一些基础服务中，注册管理中心是一个核心服务。使用注册管理中心提供的服务注册与发现的功能，就可使用智能路由、负载均衡调度和断路器等微服务治理的相关功能。

在后续章节的微服务开发和调试的过程中，我们都要启动一个注册管理中心以提供微服务治理的功能，为了节省资源使用，其他基础服务都可以设置为暂停使用的状态。



7

Rest API 微服务开发

在我们所设计的工程之中，Rest API 是一个核心的微服务应用，它包含了“domain”、“object”、“restapi”等模块的开发。其中，“domain”模块实现领域业务设计的功能，“object”模块提供了查询对象的设计，“restapi”模块对外提供高性能的接口服务。

在“domain”模块中，我们将使用领域驱动设计的方法，使用 JPA 进行实体建模、实体的持久化设计和领域服务开发。同时，还将使用缓存设计和异步消息来提高 Rest API 的访问性能。

7.1 领域业务开发

领域驱动设计（Domain-Driven Design，简称 DDD），是一种面向对象建模，并以业务模型为核心的软件开发方法。面向对象建模的设计方法，相比于面向过程和面向数据结构的设计，从根本上解耦了系统分析与系统设计之间相互隔离的状态，从而提高了软件开发的工作效率。

我们将使用 JPA 来实现领域驱动设计的开发方法。JPA 通过实体定义建立了领域业务对象的数据模型，然后通过使用存储库给实体赋予了操作行为，从而可以快速地进行领域业务功能的开发。

下面以订单微服务工程“order-microservice”的“order-domain”模块开发为例进行说明。

订单微服务工程的完整代码可以通过如下链接获得：

<https://gitee.com/chenshaojian/order-microservice.git>



首先，在模块中增加如下所示的依赖配置：

```
<dependencies>
  <dependency>
    <groupId>com.demo</groupId>
    <artifactId>order-object</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.2</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>22.0</version>
  </dependency>

  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.0</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
</dependencies>
```



这个配置主要包含如下一些组件的引用：

- 引用了 “order-object” 模块。
- 引用了 Druid 数据源组件。
- 引用了 JPA 服务组件。
- 引用了 MySQL 的驱动组件。
- 引用了 Redis 数据库组件。

怎样使用这些组件进行领域业务开发，下面将逐一进行详细说明。

7.1.1 使用 Druid 数据源

Druid 是阿里开源的一个数据源服务组件，它不但具有很好的性能，而且也提供了监控和安全过滤的功能。

如果我们要使用 Druid 监控和过滤的功能，可以增加一个配置类来启用这种功能，如下所示：

```
@Configuration
public class DruidConfiguration {
    @Bean
    public ServletRegistrationBean statViewServle(){
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new StatViewServlet(), "/druid/*");
        //IP 白名单
        servletRegistrationBean.addInitParameter("allow", "192.168.0.*,
127.0.0.1");
        //IP 黑名单 (存在共同时, deny 优先于 allow)
        servletRegistrationBean.addInitParameter("deny", "192.168.1.100");
        //控制台管理用户
        servletRegistrationBean.addInitParameter("loginUsername", "druid");
        servletRegistrationBean.addInitParameter("loginPassword",
"12345678");
        //是否能够重置数据
        servletRegistrationBean.addInitParameter("resetEnable", "false");
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean statFilter(){
        FilterRegistrationBean filterRegistrationBean = new FilterRegistration
Bean(new WebStatFilter());
```




```
//添加过滤规则
filterRegistrationBean.addUrlPatterns("//*");
//忽略过滤的格式
filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,
*.jpg,*.png,*.css,*.ico,/druid/*");
return filterRegistrationBean;
}
}
```

使用这个监控配置，当应用运行时，例如，我们启动“order-restapi”应用，即可通过如下链接打开监控控制台页面：

<http://localhost:9095/druid>

在登录认证中输入上面程序中配置的用户名和密码“druid/12345678”，即可打开如图 7-1 所示的操作界面。

注意你本地的 IP 不在上面程序设置的黑名单之中。

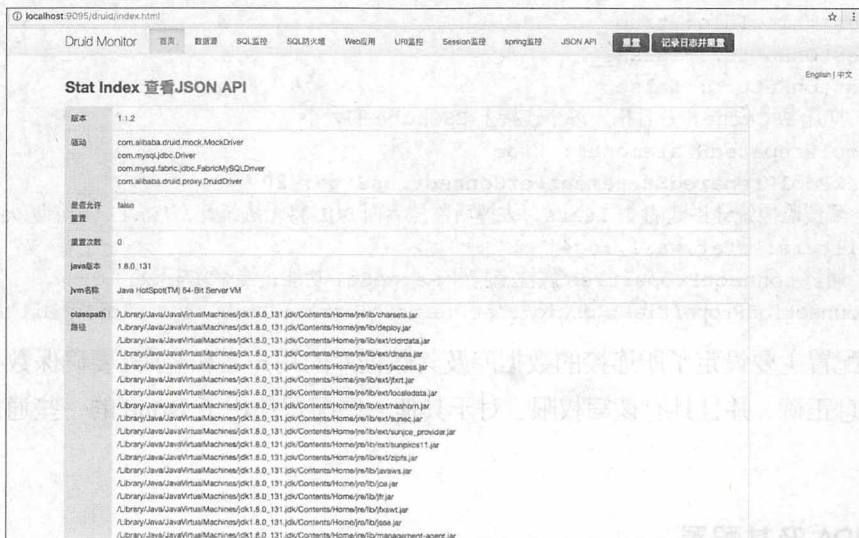


图 7-1 Druid 数据库监控

使用这个监控，通过查看“SQL 监控”的结果，可以对应用的 SQL 设计和优化提供有价值的信息。

Druid 连接数据源的配置不在“order-domain”这个模块中设置，因为这个模块将作为一个程序包提供其他应用模块使用，所以数据源的配置是在引用了这个模块的“order-restapi”模块中进行设置的，如下所示：


```

spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/orderdb?characterEncoding=utf8&useSSL=
false
    username: root
    password: 12345678
    # 初始化大小, 最小, 最大
    initialSize: 5
    minIdle: 5
    maxActive: 20
    # 配置获取连接等待超时的时间
    maxWait: 60000
    # 配置间隔多久进行一次检测, 检测需要关闭的空闲连接, 单位是毫秒
    timeBetweenEvictionRunsMillis: 60000
    # 配置一个连接在池中最小生存的时间, 单位是毫秒
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    # 打开 PSCache, 并且指定每个连接上 PSCache 的大小
    poolPreparedStatements: true
    maxPoolPreparedStatementPerConnectionSize: 20
    # 配置监控统计拦截的 filters, 去掉后监控界面 SQL 将无法统计, 'wall' 用于防火墙
    filters: stat,wall,log4j
    # 通过 connectProperties 属性来打开 mergeSql 功能; 慢 SQL 记录
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000

```

上面配置主要设定了所连接的数据库及其数据库的用户名和密码, 要确保数据库的用户配置信息正确, 并且具有读写权限。对于其他一些配置使用了 **Druid** 的一些通用参数进行设定。

7.1.2 JPA 及其配置

JPA (Java Persistence API) 是 Java 的持久层规范接口, JPA 的实现使用了 **Hibernate** 框架, 所以它的一些设计跟使用 **Hibernate** 很相似, 但 JPA 并不等同于 **Hibernate**, 它是在 **Hibernate** 之上的一个通用规范。

使用 JPA, 我们可以做到:

- 非常容易地实现实体持久化。



- 能够自动创建和更新表结构。
- 可以通过 SQL 查询语言增强程序的功能。

JPA 的配置也在引用“order-domain”的“order-restapi”模块中进行设置，如下所示：

```
spring:
  jpa:
    database: MYSQL
    show-sql: false
    ## Hibernate ddl auto (validate|create|create-drop|update)
    hibernate:
      ddl-auto: update
      #naming-strategy: org.hibernate.cfg.ImprovedNamingStrategy
      naming.physical-strategy: org.hibernate.boot.model.naming.Physical
NamingStrategyStandardImpl
      properties:
        hibernate:
          dialect: org.hibernate.dialect.MySQL5Dialect
```

其中，“ddl-auto”设置为“update”，即表示当实体属性更改时，将会更新表结构，如果表结构不存在即创建表结构。注意不要将“ddl-auto”设置为“create”或“create-drop”，那样的话，程序每次启动都会重新创建表结构，以前的数据将会丢失。如果不使用自动功能可以设置为“none”。

7.1.3 数据实体建模

使用 JPA 进行实体建模，主要使用了 Hibernate 的对象关系映射（ORM）来实现。订单服务的业务模型，由订单和订单明细两个对象所组成。ORM 主要通过注解进行数据实体设计，在注解的使用中，要注意处理好对象之间的关联关系。

实现订单实体设计的代码如下所示：

```
@Entity
@Table(name = "t_order")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="order_no",length = 64)
    private String orderNo;
    private Long userid;
    private Long merchantid;
```




```

private double amount;
@Column(name = "status", columnDefinition = "tinyint default 0")
private Integer status;
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
@Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
@Temporal(TemporalType.TIMESTAMP)
private Date created;
private String operator;
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
private Date modify;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(name = "order_detail",
    joinColumns = {@JoinColumn(name = "order_id")},
    inverseJoinColumns = {@JoinColumn(name = "detail_id")})
@JsonBackReference
private Set<OrderDetail> orderDetails = new HashSet<>();

public Order() {
}

public void addOrderDetails(OrderDetail orderDetail){
    orderDetails.add(orderDetail);
}

.....
}

```

对于这个实体的设计，主要说明以下几点重要的功能。

1. 实体对象必须有一个唯一标识

这里使用 Long 类型定义了对应的身份标识“id”，并且这个“id”将由数据库自动生成。在生产实际中，推荐使用 UUID 作为对象的唯一标识，这样做的好处是，不但可以保持这一字段长度的一致性，还能保证这一标识在整个数据库中的唯一性，这样非常有利于数据库的集群设计。

2. 日期属性要使用正确的格式

日期使用注解“@DateTimeFormat”进行格式化，可以保证日期的显示和在参数传递中的正确性。注意上面的两个日期中创建日期“created”使用了默认值，而修改日期“modify”并没有使用默认值。



3. 注意属性的大小写在数据库中的表现形式

例如，属性“orderNo”在数据库中最好设置为“order_no”，所以如果使用了注解“@Column”来设置数据库的字段必须要遵循这种规则。

4. 使用合理的关联设置

关联设置是实体设计的关键，为了避免引起递归调用，最好都使用单向关联设置，即在互相关联的两个对象之中，只在一个主对象中进行关联设置就可以了。另外，为了避免递归调用，还可以在关联中使用注解“@JsonBackReference”或“@JsonIgnore”进行配置。

订单实体使用了注解“@OneToMany”来关联订单明细对象，并使用了一个中间表的方式来保存这种关联关系。一般地，一对多和多对多的关联可以使用中间表来存储关联关系，而多对一的关联关系可以使用一个字段来存储关联对象的外键。

在订单实体的关联设置中，我们还使用了级联的操作设置“CascadeType.ALL”。这样，在订单实体中的所有操作都将影响到订单明细的操作。例如，执行订单保存时，订单明细也会自动得到保存。

级联的设置可以使用如下所示的配置。

- CascadeType.PERSIST：级联保存。
- CascadeType.REMOVE：级联删除。
- CascadeType.MERGE：级联合并（更新）。
- CascadeType.DETACH：级联脱管/游离。
- CascadeType.REFRESH：级联刷新。
- CascadeType.ALL：以上所有级联操作。

订单明细的实体设计如下所示：

```
@Entity
@Table(name = "t_orderdetail")
public class OrderDetail {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long goodsid;
    private String goodsname;
    private String photo;
```



```
private Integer nums;
private float price;
private double money;
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
@Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
@Temporal(TemporalType.TIMESTAMP)
private Date created;

public OrderDetail() {
}
.....
}
```

订单明细保存的是商品信息，这里使用了冗余设计的方法，既保存了商品标识，也保存了商品名称和图片等冗余数据。因为对于一般的查询，只要显示商品名称和商品的图片就足够了。如果要进行更加详细的查询，可以使用商品标识通过接口进行调用，即通过商品服务的接口来查询商品。因此，使用冗余设计也是可以提高订单访问性能的一种方法。

7.1.4 查询对象设计

我们将查询对象设计放在一个模块之中，这样，可以在其他模块或者其他工程中进行调用。使用查询对象（Query Object）即 `qo`，是为了与 `vo` 进行区分。有人把 `vo` 看作值对象（Value Object），也有人把 `vo` 看作视图对象（View Object），所以很容易引起误解。这两种对象的意义和用途是不一样的，值对象表示与实体不同的一些数据，它也可以作为视图显示，而视图对象只是作为视图显示的一种数据。

因为实体是有生命周期和状态的，它的状态改变将会直接影响到存储的数据，所以我们使用一个无状态的数据对象来存取实体的数据，这些数据的使用和更改将不会直接影响到数据存储，因此也可以用之即弃。

使用查询对象，我们既可以将它作为值对象使用，也可以作为视图对象使用，而且还可以作为查询参数的一个集合来使用，即相当于一个数据传输对象（Data Transfer Object，简称 `dto`）。

我们只要使用一个查询对象 `qo`，就可以包含 `vo`、`dto` 等对象的功能，这是一种简化设计。`qo` 有时包含了一些冗余数据，但这对于使用方来说没有什么影响，使用方只关心自己需要的数据，而不关心与自己无关的数据。例如，在查询对象中，将会包含分页所需要



用到的页码和页大小等分页属性数据，而在视图显示中并不需要这些数据，它可以不用去理会这些数据。

订单的查询对象设计在“order-object”模块之中实现。

订单查询对象的设计参照了订单实体的属性设计，实现代码如下：

```
public class OrderQo extends PageQo{
    private Long id;
    private String orderNo;
    private Long userid;
    private Long merchantid;
    private double amount;
    private Integer status;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date created;
    private String operator;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date modify;

    private Set<OrderDetailQo> orderDetails = new HashSet<>();

    public OrderQo() {
    }
    .....
}
```

订单查询对象与实体的不同之处在于没有那些与数据相互关联的注解，而多了几个分页的属性字段。根据实际需要，还可以增加其他一些属性字段。

其中，分页的属性设计在页对象中进行定义，它的设计如下所示：

```
package com.demo.order.object;

public class PageQo {
    private Integer page = 0;
    private Integer size = 10;

    public Integer getPage() {
        return page;
    }

    public void setPage(Integer page) {
        this.page = page;
    }
}
```



```

    public Integer getSize() {
        return size;
    }

    public void setSize(Integer size) {
        this.size = size;
    }
}

```

7.1.5 实体持久化设计

使用 JPA 进行实体的持久化设计是非常简单的，只要为实体创建一个存储库接口，将实体对象与 JPA 的存储库接口进行绑定，就可以实现实体的持久化设计，这相当于给实体赋予了一些访问数据库的操作行为。

例如，对于订单实体，我们只要创建一个如下所示的存储库接口，就可以使用这个接口执行创建订单、删除订单和查询订单的一系列行为。

```

@Repository
public interface OrderRepository extends JpaRepository<Order, Long>{
}

```

所以，使用 JPA 可以非常轻易地实现实体持久化，不用编写任何 SQL 查询语句就可以操作数据库。

当然，在一个数据服务应用中，由于业务功能的复杂性，做到完全没有查询语句是不大可能的。对于上面这个存储库接口，我们还可以使用查询语句来增强它的功能，如下所示：

```

@Repository
public interface OrderRepository extends JpaRepository<Order, Long>,
JpaSpecificationExecutor<Order> {
    Page<Order> findByCreatedGreaterThan(@Param("created") Date created,
Pageable pageRequest);

    @Modifying
    @Query("delete from Order t where t.userid = ?1")
    void deleteByUserid(Long userid);

    @Query("select o from Order o " +
        "left join o.orderDetails d " +
        "where d.id = :id")
}

```




```
Order findById(@Param("id") Long id);
}
```

这里，第一个查询语句可以使用“userid”删除订单；第二个查询语句的定义，可以使用订单明细的标识来查询订单。与一般查询语句不同的是，这里的查询语句使用实体对象来表示表名，JPA 将根据实体对象的定义将对象名称转化为表名。

另外，在上面的存储库接口设计中，还使用多重继承的方法，继承了一个具有分页功能的接口“JpaSpecificationExecutor”，这为后面功能更加强大的分页设计做好了准备。

这里也使用了一个简单的分页设计，即上面的声明方法“findByCreatedGreaterThan”实现了按创建日期参数进行分页查询的功能。

7.1.6 持久化测试

完成实体的持久化设计之后，我们就可以对持久化设计进行一个测试，以验证这些设计的正确性。测试可以在“order-domain”模块中进行，因为这个模块最终是作为程序包的方式对外提供服务的，所以这里并没有数据源和 JPA 相关的一些配置，为此创建了一个配置类：“JpaConfiguration.java”用来进行测试。其中，数据源和 JPA 的配置与上面的配置是一样的。如下所示是其中数据源的配置：

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/orderdb?character
Encoding=utf8");
    dataSource.setUsername("root");
    dataSource.setPassword("12345678");

    return dataSource;
}
```

如果还没有创建“orderdb”数据库，必须在所连接的 MySQL 中进行创建。

这样，我们就可以使用下面的测试用例进行测试：

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {JpaConfiguration.class})
public class TestDb {
    @Autowired
    private OrderRepository orderRepository;
```



```

@Test
public void insertData(){
    OrderDetail orderDetail = new OrderDetail();
    orderDetail.setGoodsname("测试商品 1");
    orderDetail.setGoodsid(1L);
    orderDetail.setPrice(12.20F);
    orderDetail.setNums(1);
    orderDetail.setMoney(12.20D);
    orderDetail.setPhoto("../images/order/orderPic.jpg");

    OrderDetail orderDetail2 = new OrderDetail();
    orderDetail2.setGoodsname("测试商品 2");
    orderDetail2.setGoodsid(2L);
    orderDetail2.setPrice(20.00F);
    orderDetail2.setNums(2);
    orderDetail2.setMoney(40.00D);
    orderDetail2.setPhoto("../images/order/orderPic.jpg");

    Order order = new Order();
    order.setOrderNo("20170930000003");
    order.setUserid(11111235L);
    order.setAmount(52.20D);
    order.setStatus(2);
    order.addOrderDetails(orderDetail);
    order.addOrderDetails(orderDetail2);
    orderRepository.save(order);
    Assert.notNull(order.getId(), "not insert");
}
}

```

在这个测试中创建了一个订单，并且它具有两个明细记录。

如果测试通过，将在数据库“orderdb”中创建与实体设计相对应的表结构，然后写入订单和明细记录数据。在这个测试用例中我们并没有编写保存订单明细的操作行为，但是由于实体关联的级联作用，当进行订单保存时，订单明细也相应地自动保存了。

我们还可以进行读取数据、更改数据和删除数据的测试，以验证上面设计的正确性。如下所示是一个查询列表数据的测试：

```

@Test
public void getData(){
    List<Order> orders = orderRepository.findAll();
    Assert.notEmpty(orders, "list empty");
    for(Order order : orders) {

```




```

        logger.info("=====order id={}, order no={}, goods name={}",
                    order.getId(), order.getOrderNo(), order.getOrderDetails().
iterator().next().getGoodsname());
    }
}

```

在这个测试用例中，我们在断言中设定了当列表为空时就显示“list empty”信息，否则打印出数据的简要信息。

如果所有测试无误，说明上面的一些设计是正确的，可以继续下面的开发，否则可根据测试时出现的问题进行相关的修改和调整，直到全部测试通过为止。

7.1.7 领域服务开发

在领域业务开发中，最终并不是以存储库接口对外提供服务的，而是使用一个服务层进行封装，然后对外提供领域服务。这样做，还具有如下的好处：

- 有利于在一个服务中调用多个存储库接口时，使用统一的事务管理。
- 方便在一个服务中增强功能设计，例如，增加缓存和查询分页的设计等。

如下所示代码是订单服务“OrderService”的开发实例：

```

@Service
@Transactional
public class OrderService {
    @Autowired
    private OrderRepository orderRepository;

    public Order findOne(Long id){
        return orderRepository.findOne(id);
    }

    public void save(Order order){
        orderRepository.save(order);
    }

    public void delete(Long id){
        orderRepository.delete(id);
    }

    public Page<Order> findAll(OrderQo orderQo){
        Sort sort = new Sort(Sort.Direction.DISC, "created");
    }
}

```



```

        Pageable pageable = new PageRequest(orderQo.getPage(), orderQo.
getSize(), sort);

        return orderRepository.findAll(new Specification<Order>(){
            @Override
            public Predicate toPredicate(Root<Order> root, CriteriaQuery<?>
query, CriteriaBuilder criteriaBuilder) {
                List<Predicate> predicatesList = new ArrayList<Predicate>();

                if(CommonUtils.isNotNull(orderQo.getUserid())) {
                    predicatesList.add(criteriaBuilder.equal(root.get("userid"),
orderQo.getUserid()));
                }
                if(CommonUtils.isNotNull(orderQo.getMerchantid())) {
                    predicatesList.add(criteriaBuilder.equal(root.get
("merchantid"), orderQo.getMerchantid()));
                }
                if(CommonUtils.isNotNull(orderQo.getOrderNo())) {
                    predicatesList.add(criteriaBuilder.equal(root.get
("orderNo"), orderQo.getOrderNo()));
                }
                if(CommonUtils.isNotNull(orderQo.getStatus())) {
                    predicatesList.add(criteriaBuilder.equal(root.get
("status"), orderQo.getStatus()));
                }
                if(CommonUtils.isNotNull(orderQo.getCreated())){
                    predicatesList.add(criteriaBuilder.greaterThan(root.get
("created"), orderQo.getCreated()));
                }

                query.where(predicatesList.toArray(new Predicate[predicates
List.size()]));

                return query.getRestriction();
            }
        }, pageable);
    }
}

```

在这个设计中，主要实现了如下一些功能：

- 使用注解 “@Transactional” 为订单服务实现统一的隐式事务管理。
- 通过调用 “OrderRepository” 存储库接口，提供数据的查询、保存和删除等操作。



- 使用一个增强功能的分页查询设计。在这个设计中，通过使用 JPA 的 “findAll” 方法，并根据查询对象 “OrderQo” 传递的参数，实现可以根据参数进行智能判断，灵活地根据提供的查询参数执行相关的 SQL 查询。

7.1.8 领域服务的单元测试

完成领域服务设计之后，我们可以在 “order-restapi” 模块中写一个完整的单元测试来验证上面领域服务的设计。需要注意的是，这里的测试中有关数据源和 JPA 的配置，都使用了模块的配置文件 “application.yml” 中的配置。通过单元测试之后，就可以在 “order-restapi” 模块中正常使用订单服务。单元测试可以包含实体对象的创建、查询、修改和更新等行为。下面的测试用例演示了如何生成一个订单，然后对订单数据使用分页查询的方法：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {JpaConfiguration.class, OrderRestApi
Application.class})
@SpringBootTest
public class OrderTest {
    private static Logger logger = LoggerFactory.getLogger(OrderTest.class);
    @Autowired
    private OrderService ordersService;

    @Before
    public void insertData(){
        OrderDetail orderDetail = new OrderDetail();
        orderDetail.setGoodsname("测试商品 1");
        orderDetail.setGoodsid(1L);
        orderDetail.setPrice(12.20F);
        orderDetail.setNums(1);
        orderDetail.setMoney(12.20D);
        orderDetail.setPhoto("../images/order/orderPic.jpg");

        Order order = new Order();
        order.setOrderNo("201709300000004");
        order.setUserid(11111235L);
        order.setAmount(52.20D);
        order.setStatus(1);
        order.addOrderDetails(orderDetail);
        ordersService.save(order);
        Assert.assertNotNull(order.getId(), "not insert");
    }
}
```



```

    }

    @Test
    public void findAll() throws Exception{
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = sdf.parse("2017-01-01 00:00:00");
        OrderQo orderQo = new OrderQo();
        Page<Order> page = ordersService.findAll(orderQo);

        Assert.notEmpty(page.getContent(), "list is empty");

        for(Order order : page.getContent()) {
            logger.info("=====order id={}, order userid={}, detail goods
name = {}",
                        order.getId(), order.getUserid(), order.getOrderDetails().
iterator().next().getGoodsname());
        }
    }
}

```

在测试中我们使用断言 `Assert` 对结果进行判断，如果断言的表达式结果为真，则测试成功通过。其中，对于分页查询，可以通过上面代码调整各种查询参数进行测试，测试通过都会输出查询结果的简要信息。

有关更新和删除的方法也可以参照上面的设计实现，详细情况可以参考源代码 `OrderTest.java` 的设计。

需要注意的是，因为“`order-restapi`”模块使用了消息总线的功能，如果没有连接 `RabbitMQ` 服务器，测试过程中将会有一些错误提示，但并不影响领域服务的测试。有关消息总线的配置，请参照后面 7.3 节的说明。

7.1.9 使用 Redis 实现缓存设计

缓存设计，我们首选 `Redis`。`Redis` 是一个高性能的 NoSQL 数据库，具有非常高效的存取性能，并且具有很好的扩展性。`Redis` 首先使用内存作为数据存取的缓冲区，同时也能将数据持久地保存在磁盘中。使用 `Redis`，必须先进行服务器的安装和配置，有关 `Redis` 的安装方法请参考第 14 章的说明。

使用缓存，不但能提高数据的读取性能，也是我们解决数据库集群设计中数据同步延迟问题的最好方法。在数据库的集群中使用读写分离的设计时，数据库同步总免不了会出



现些许延迟的情况,这种情况在数据库管理系统本身并没有很好的解决方法,所以最好的方法就是使用缓存来处理。

在本书的实例中,商家服务工程“merchant-microservice”已经实现了缓存的设计,可以通过下列链接获得完整的源代码:

```
https://gitee.com/chenshaojian/merchant-microservice.git
```

在商家服务工程“merchant-microservice”中,缓存设计通过下列步骤来实现。

首先,在商家服务工程的“merchant-domain”模块中,通过项目对象模型即 pom.xml 增加 Redis 的依赖配置,如下所示:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

然后,在引用“merchant-domain”的“merchant-restapi”模块中,通过应用配置文件“application.yml”,设置如下所示的 Redis 服务器连接参数:

```
spring:
  redis:
    host: 127.0.0.1
    port: 6379
    pool:
      max-idle: 8
      min-idle: 0
      max-active: 8
      max-wait: -1
```

其中,“host”和“port”的参数根据 Redis 服务器的安装情况进行配置。

在 Spring 框架中,使用 Redis 是非常简单的,通过 RedisTemplate 就能实现各种数据存取操作。

为了方便程序的调用,我们设计了一个 CacheComponent 组件来使用 RedisTemplate,实现代码如下所示:

```
@Component
public class CacheComponent {
    @Resource(name = "redisTemplate")
    private RedisTemplate<String, Object> redisTemplate;

    public void put(String key, String childKey, Object object, long timeout) {
```



```
        putForExpire(key, childKey, object, timeout);
    }

    public Object get(String key, String childKey) {
        return getForExpire(key, childKey);
    }

    public void remove(String key, String childKey) {
        removeForExpire(key, childKey);
    }

    public void putForExpire(String key, String childKey, Object value, long
timeout) {
        BoundValueOperations<String, Object> boundValueOperations =
redisTemplate.boundValueOps(getFinallyKey(key, childKey));
        boundValueOperations.set(value, timeout, TimeUnit.SECONDS);
    }

    public Object getForExpire(String key, String childKey) {
        return redisTemplate.boundValueOps(getFinallyKey(key, childKey)).get();
    }

    public void removeForExpire(String key, String childKey) {
        redisTemplate.delete(getFinallyKey(key, childKey));
    }

    private String getFinallyKey(String key, String childKey) {
        return key + childKey;
    }
}
```

在这个组件设计中，通过提供如下几个通用方法实现在 Redis 中存取数据的操作。

- get: 读取数据
- put: 写入数据
- remove: 删除数据

其中，为了提高 Key 的可读性和唯一性，我们使用了主键（key）和子键（childKey）来组成一个唯一的 key。另外，在使用 put 方法时，以秒为单位来设定缓存数据的有效期。这样我们就可以在访问数据库的程序中，通过调用 CacheComponent 组件来实现使用缓存的设计。

例如，在“merchant-domain”模块的领域服务设计中，其中商家用户的领域服务使用



缓存的方法如下所示:

```
@Service
@Transactional
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private CacheComponent cacheComponent;

    public void save(User user){
        //删除缓存
        if(CommonUtils.isNotNull(user.getId())){
            String key = user.getId().toString();
            cacheComponent.remove(Constant.MERCHANT_CENTER_USER_ID, key);
        }
        //删除原有缓存
        userRepository.save(user);
        //保存缓存
        if(CommonUtils.isNotNull(user.getId())){
            String key = user.getId().toString();
            cacheComponent.put(Constant.MERCHANT_CENTER_USER_ID, key, user,
12); //增加缓存, 保存 12 秒
        }
    }

    public void delete(Long id){
        //删除缓存
        cacheComponent.remove(Constant.MERCHANT_CENTER_USER_ID, id.toString());
        userRepository.delete(id);
    }

    public List<User> findAll(){
        return userRepository.findAll();
    }

    public User findOne(Long id){
        User user = null;
        //使用缓存
        Object object = cacheComponent.get(Constant.MERCHANT_CENTER_USER_ID,
id.toString());
        if (CommonUtils.isNull(object)) {
            user = userRepository.findById(id);
            if (user != null)
```



```
        cacheComponent.put(Constant.MERCHANT_CENTER_USER_ID,
id.toString(), user, 12);
    } else {
        user = (User) object;
    }
    return user;
}
}
```

这个程序对缓存使用的设计主要体现了如下几点：

- 在数据保存或更新时同时更新缓存。
- 删除数据时首先删除缓存。
- 读取数据时先查询缓存，如果缓存不存在再从数据库中读取，并同时保存缓存。

其中，缓存的有效期设置为 12 秒。这样设置的目的是：一方面为了提高 Redis 的访问性能，提高缓存的命中率；另一方面，在这个时间之内，已经足够让数据库管理系统的主从数据库进行数据同步。

这样，当程序访问上面方法时，就会使用缓存来暂存数据，当我们执行用户创建、更新或者获取一个用户对象时，就可以在 12 秒内从 Redis 服务器中查询到缓存数据。

例如，当程序中调用了用户领域服务的 `findOne` 方法时，在 Redis 服务器中通过使用“redis-cli”来查看缓存的使用情况，执行情况如下所示：

```
sh-3.2# redis-cli
127.0.0.1:6379> keys *
1) "foo"
2) "\xac\xed\x00\x05t\x00\x19MERCHANT_CENTER_USER_ID_1"
127.0.0.1:6379> keys *
1) "foo"
127.0.0.1:6379> quit
```

缓存的读写与数据库的读写是一起执行的，为了保证在缓存中不会读到脏数据，并且提高缓存的命中率，缓存设计必须遵循如下的原则：

- 在写入数据时，先写数据库，再写缓存。
- 在更新数据时，先删除缓存，再更新数据和保存缓存。
- 在删除数据时，先删除缓存，再在数据库中删除数据。
- 在读取数据时，先读缓存，当缓存不存在时，再读数据库，同时也将数据写入缓存之中。



7.2 Rest API 应用开发

完成“domain”模块的领域服务开发之后,就可以进行 Rest API 微服务应用的开发了。

“domain”模块只为 Rest API 微服务所专用,并不提供给其他 Web UI 应用使用,Web UI 应用所使用的数据通过调用 Rest API 微服务取得。另外,为了保证 Rest API 的独立性,在 Rest API 之间也不进行互相调用。如果 Rest API 之间需要通信,可以通过异步消息来实现。

Rest API 是一个独立的 Web 应用,可以独立部署、独立运行,并且使用了独立的数据库管理系统。Rest API 还可以根据需要进行多实例部署,然后,通过微服务治理提供高性能的服务。

下面使用订单服务工程即“order-microservice”来说明如何进行 Rest API 的开发。订单服务的 Rest API 微服务的开发在模块“order-restapi”中进行。

7.2.1 Rest API 应用配置

首先,在“order-restapi”模块的依赖配置中使用如下所示的设置:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <!--消息总线-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>com.demo</groupId>
    <artifactId>order-domain</artifactId>
```



```

        <version>${project.version}</version>
    </dependency>
</dependencies>

```

这是模块的核心配置，包含如下一些功能组件：

- 智能路由组件 Zuul，它包含负载均衡组件 Ribbon。
- 注册管理服务组件 Eureka，提供服务注册和发现的功能。
- “bus-amqp” 消息服务总线。
- Web 服务组件，提供 Web 应用的功能。
- “order-domain” 模块引用。

为了方便调试，这里省略了配置管理服务、跟踪服务、日志服务等依赖引用配置，这些配置在源代码中也已经进行了注释，如果要启用相关的功能，删除掉注释标志即可使用。

在模块的配置文件 “application.yml” 中使用如下所示的设置：

```

server:
  port: 9095

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    preferIpAddress: true

spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/orderdb?characterEncoding=utf8&useSSL=
false
    username: root
    password: 12345678

  jpa:
    database: MYSQL
    show-sql: false
    hibernate:
      ddl-auto: update
      naming.physical-strategy:
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

```




```
properties:
  hibernate:
    dialect: org.hibernate.dialect.MySQL5Dialect
```

这些配置设定了服务端口，连接注册中心的参数和数据源的参数等，其中，连接数据源和使用 JPA 的配置，是引用“order-domain”所需要的配置。

别忘了一个非常重要的配置，即应用的名称，这将作为这个应用在注册服务中心的一个唯一标识，这个配置在“bootstrap.yml”文件之中设置，如下所示将应用名称设为“orderapi”：

```
spring:
  application:
    name: orderapi
```

7.2.2 启动程序设计

Rest API 应用的启动程序为“OrderRestApiApplication”，如下所示：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
@ComponentScan(basePackages = "com.demo")
public class OrderRestApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderRestApiApplication.class, args);
    }
}
```

这里除了具有一般 Spring Boot 的启动程序的功能，还增加服务发现客户端和路由代理的功能，分别使用注解“@EnableDiscoveryClient”和“@EnableZuulProxy”来实现，而注解“@ComponentScan”可以保证“order-domain”模块的组件能够被正常扫描并且加载进来。

7.2.3 接口开发

对于订单服务来说，我们遵循 Rest 协议，提供下列几种接口的开发。

- GET /order/{id}：获取一个订单的详细信息。
- GET /order：查询订单的分页信息。
- POST /order：创建一个新订单。



- PUT /order: 更新一个订单。
- DELETE /order/{id}: 删除一个订单。

获取一个订单的详细信息使用 GET 方法，完成设计的代码如下所示：

```
@RequestMapping(value="/{id}")
public CompletableFuture<String> findById(@PathVariable Long id) {
    return CompletableFuture.supplyAsync(() -> {
        Order order = orderService.findOne(id);
        return new Gson().toJson(order);
    });
}
```

其中，“CompletableFuture”是 Java 的非阻塞异步编程方法，因为在 Web UI 微服务对 Rest API 的调用中将使用这种高并发的编程方法，所以为了保证与调用者保持同步，这里也使用了这种方法。有关非阻塞异步编程方法的设计将在下一章中进行详细的说明。

订单的分页查询同样也是使用 GET 方法。其中，对分页的查询参数的处理实现了根据输入参数进行智能判断的设计，参数非空时才使用该参数进行查询。完成设计的代码如下所示：

```
@RequestMapping(method = RequestMethod.GET)
public CompletableFuture<String> findAll(Integer index, Integer size, Long
userid, Long merchantid, Integer status, String created) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            OrderQo orderQo = new OrderQo();

            if (CommonUtils.isNotNull(index)) {
                orderQo.setPage(index);
            }
            if (CommonUtils.isNotNull(size)) {
                orderQo.setSize(size);
            }
            if (CommonUtils.isNotNull(userid)) {
                orderQo.setUserid(userid);
            }
            if (CommonUtils.isNotNull(merchantid)) {
                orderQo.setMerchantid(merchantid);
            }
            if (CommonUtils.isNotNull(status)) {
                orderQo.setStatus(status);
            }
        }
    });
}
```




```

        if(CommonUtils.isNotNull(created)){
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
            orderQo.setCreated(sdf.parse(created));
        }

        Page<Order> orderses = orderService.findAll(orderQo);

        Map<String, Object> page = new HashMap<>();
        page.put("content", orderses.getContent());
        page.put("totalPages", orderses.getTotalPages());
        page.put("totalelements", orderses.getTotalElements());

        return new Gson().toJson(page);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
});
}

```

在所有查询参数中，涉及分页的页码、页大小等参数将不能为空，如果调用方没有提供这些参数，程序将使用默认值，其他几个参数（例如，商家 ID、用户 ID、订单状态等）都可作为可选参数。

创建订单的设计使用 POST 方法，可以接收一个完整的查询对象作为输入参数，实现的代码如下所示：

```

@RequestMapping(method = RequestMethod.POST)
public CompletableFuture<String> save(@RequestBody OrderQo orderQo)
throws Exception{
    return CompletableFuture.supplyAsync(() -> {
        Order order = CopyUtil.copy(orderQo, Order.class);

        List<OrderDetail> detailList = CopyUtil.copyList(orderQo.
getOrderDetails(), OrderDetail.class);
        order.setOrderDetails(detailList);

        orderService.save(order);

        logger.info("新增->ID=" + order.getId());
        return order.getId().toString();
    });
}

```



这里，使用“@RequestBody”来接收对象输入，因为输入的是查询对象，所以需要将查询对象转换成实体对象后再进行保存。如果对象中有关联的子对象，每个子对象同样也要进行转换。

订单更新的设计使用 PUT 方法，实现的代码如下所示：

```
@RequestMapping(method = RequestMethod.PUT)
public CompletableFuture<String> update(@RequestBody OrderQo orderQo)
throws Exception{
    return CompletableFuture.supplyAsync(() -> {
        Order order = CopyUtil.copy(orderQo, Order.class);
        order.setModify(new Date());

        List<OrderDetail> detailList = CopyUtil.copyList(orderQo.getOrder
Details(), OrderDetail.class);
        order.setOrderDetails(detailList);

        orderService.save(order);

        logger.info("修改->ID=" + order.getId());
        return order.getId().toString();
    });
}
```

其中，数据对象的处理与创建订单类似，必须将查询对象转换为实体对象。

删除订单的设计使用 DELETE 方法，实现的代码如下所示：

```
@RequestMapping(value="/{id}",method = RequestMethod.DELETE)
public CompletableFuture<String> delete(@PathVariable Long id) throws
Exception {
    return CompletableFuture.supplyAsync(() -> {
        orderService.delete(id);

        logger.info("删除->ID=" + id);
        return id.toString();
    });
}
```

删除订单时根据订单实体的级联设计，订单明细将被同时删除。

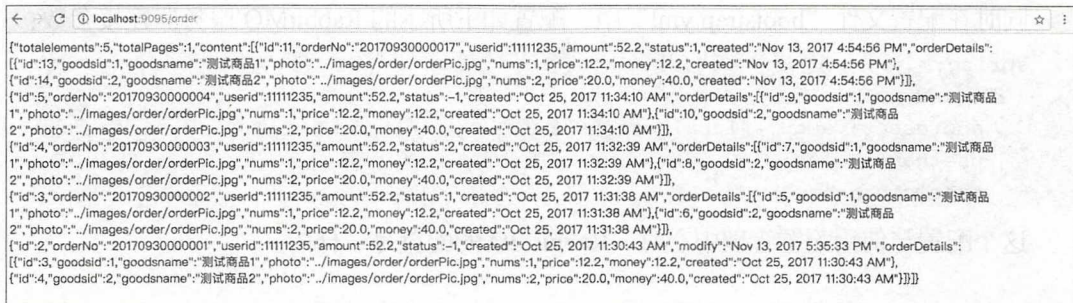
完成这些设计之后，就可以启动应用进行测试。

启动成功之后，在浏览器中输入如下链接可以查询订单的分页数据：

```
http://localhost:9095/order
```



如果数据库中具有测试数据,则可返回类似于如图 7-2 所示的分页数据,注意这些数据是以 Json 格式输出的。



```
{
  "totalElements": 5,
  "totalPages": 1,
  "content": [
    {
      "id": 11,
      "orderNo": "20170930000017",
      "userId": 11111235,
      "amount": 52.2,
      "status": 1,
      "created": "Nov 13, 2017 4:54:56 PM",
      "orderDetails": [
        {
          "id": 13,
          "goodsid": 1,
          "goodsname": "测试商品1",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 1,
          "price": 12.2,
          "money": 12.2,
          "created": "Nov 13, 2017 4:54:56 PM"
        },
        {
          "id": 14,
          "goodsid": 2,
          "goodsname": "测试商品2",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 2,
          "price": 20.0,
          "money": 40.0,
          "created": "Nov 13, 2017 4:54:56 PM"
        }
      ]
    },
    {
      "id": 5,
      "orderNo": "20170930000004",
      "userId": 11111235,
      "amount": 52.2,
      "status": -1,
      "created": "Oct 25, 2017 11:34:10 AM",
      "orderDetails": [
        {
          "id": 9,
          "goodsid": 1,
          "goodsname": "测试商品1",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 1,
          "price": 12.2,
          "money": 12.2,
          "created": "Oct 25, 2017 11:34:10 AM"
        },
        {
          "id": 2,
          "photo": ".../images/order/orderPic.jpg",
          "nums": 2,
          "price": 20.0,
          "money": 40.0,
          "created": "Oct 25, 2017 11:34:10 AM"
        }
      ]
    },
    {
      "id": 4,
      "orderNo": "20170930000003",
      "userId": 11111235,
      "amount": 52.2,
      "status": 2,
      "created": "Oct 25, 2017 11:32:39 AM",
      "orderDetails": [
        {
          "id": 7,
          "goodsid": 1,
          "goodsname": "测试商品1",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 1,
          "price": 12.2,
          "money": 12.2,
          "created": "Oct 25, 2017 11:32:39 AM"
        },
        {
          "id": 8,
          "goodsid": 2,
          "goodsname": "测试商品2",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 2,
          "price": 20.0,
          "money": 40.0,
          "created": "Oct 25, 2017 11:32:39 AM"
        }
      ]
    },
    {
      "id": 3,
      "orderNo": "20170930000002",
      "userId": 11111235,
      "amount": 52.2,
      "status": 1,
      "created": "Oct 25, 2017 11:31:38 AM",
      "orderDetails": [
        {
          "id": 5,
          "goodsid": 1,
          "goodsname": "测试商品1",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 1,
          "price": 12.2,
          "money": 12.2,
          "created": "Oct 25, 2017 11:31:38 AM"
        },
        {
          "id": 6,
          "goodsid": 2,
          "goodsname": "测试商品2",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 2,
          "price": 20.0,
          "money": 40.0,
          "created": "Oct 25, 2017 11:31:38 AM"
        }
      ]
    },
    {
      "id": 2,
      "orderNo": "20170930000001",
      "userId": 11111235,
      "amount": 52.2,
      "status": -1,
      "created": "Oct 25, 2017 11:30:43 AM",
      "modify": "Nov 13, 2017 5:35:33 PM",
      "orderDetails": [
        {
          "id": 3,
          "goodsid": 1,
          "goodsname": "测试商品1",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 1,
          "price": 12.2,
          "money": 12.2,
          "created": "Oct 25, 2017 11:30:43 AM"
        },
        {
          "id": 4,
          "goodsid": 2,
          "goodsname": "测试商品2",
          "photo": ".../images/order/orderPic.jpg",
          "nums": 2,
          "price": 20.0,
          "money": 40.0,
          "created": "Oct 25, 2017 11:30:43 AM"
        }
      ]
    }
  ]
}
```

图 7-2 订单分页查询接口测试

7.3 使用消息处理事件

使用微服务架构设计,由于数据库分布在不同的微服务之中,所以一个平台的数据同步不能像整体架构设计那样,可以在一个事务中处理,使用强一致性设计。但是我们可以依据 CAP 理论,使用最终一致性的原则来处理数据的同步。

除了在调用端可以通过调用不同的 Rest API 做到实时处理数据的同步操作之外,也可以在 Rest API 中使用异步消息来发布数据同步的事件,实现数据的最终一致性设计。

使用消息来传递事件变化的方法有很多,这里将通过与配置管理中心一样的消息总线来使用 RabbitMQ 消息通道,这种方法简单、轻量,易于实现。

消息具有生产者和消费者两个主体,消息生产者向消息通道发布消息,消息消费者从消息通道中订阅消息,从而完成一个通信流程。

下面以订单撤销这一事件为例来演示消息的使用方法。在订单服务中进行订单撤销时,在商品服务中必须减掉订单所属商品的购买数量。

需要注意的是,要进行消息通信测试,还要安装 RabbitMQ 服务器,安装方法可以参考第 14 章中的说明。

首先,在消息生产者和消费者所在的模块(即“order-restapi”和“goods-restapi”模块)中,都必须开启下列这个引用:

```
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

同时在配置文件“bootstrap.yml”中，配置如下所示的 RabbitMQ 服务器连接参数：

```
spring:
  rabbitmq:
    addresses: amqp://localhost:5672
    username: alan
    password: alan
```

这个配置与使用配置管理中心时，RabbitMQ 的配置是一样的。

7.3.1 消息生产者设计

在“order-restapi”模块中创建一个订单通道“orderschannel”，可以用来发布订单变更的消息，如下所示：

```
package com.demo.order.restapi.mqchannel;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface OutputSource {
    String ORDERSCHANNEL = "orderschannel";

    @Output("orderschannel")
    MessageChannel ordersOutput();
}
```

然后，在订单修改的控制器设计中加入消息发送的功能，即当订单修改时，把订单对象作为消息内容向订单消息通道发布消息，如下所示：

```
@RestController
@RequestMapping("/order")
@EnableBinding(OutputSource.class)
public class OrderRestController {
    private static Logger logger = LoggerFactory.getLogger(
        OrderRestController.class);

    @Autowired
    private OrderService orderService;

    @Autowired
```




```

@Output(OutputSource.ORDERSCHANNEL)
private MessageChannel ordersChannel;

@RequestMapping(method = RequestMethod.PUT)
public CompletableFuture<String> update(@RequestBody OrderQo orderQo)
throws Exception{
    return CompletableFuture.supplyAsync(() -> {
        Order order = CopyUtil.copy(orderQo, Order.class);
        order.setModify(new Date());

        List<OrderDetail> detailList = CopyUtil.copyList(orderQo.
getOrderDetails(), OrderDetail.class);
        order.setOrderDetails(detailList);

        orderService.save(order);

        //发送 MQ 消息, 通知订单修改
        ordersChannel.send(MessageBuilder.withPayload(orderQo).build());

        logger.info("修改->ID=" + order.getId());
        return order.getId().toString();
    });
}

```

实现代码中首先使用“@EnableBinding”绑定了订单消息通道, 然后使用“MessageChannel”初始化一个订单消息通道“ordersChannel”, 最后使用这个消息通道的“send”方法则可以发布消息。

7.3.2 消息消费者设计

为了在商品服务工程之中可以使用订单服务工程的查询对象, 我们先使用下列方法将订单服务的查询对象进行打包, 然后发布到 Maven 的组件仓库之中。

首先, 在订单服务工程“order-microservice”中将项目打包, 发布在本地的 Maven 的组件仓库中。操作方法如图 7-3 所示, 在 IDEA 中, 通过打开右侧边栏的“Maven Projects”, 展开项目的“Lifecycle”, 然后双击“install”即可执行组件的安装。操作完成之后, 在其他项目工程中, 就可以引用这个项目的组件了。



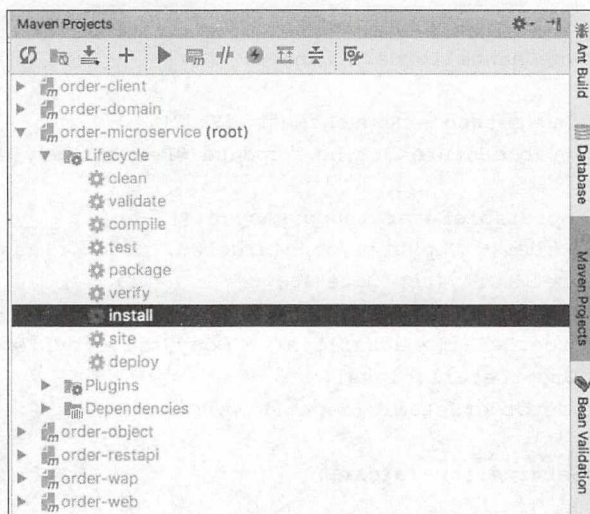


图 7-3 安装组件到本地的 Maven 仓库

然后，在商品服务工程增加如下所示的依赖引用，即引用了订单服务的查询对象“order-object”的程序包：

```
<dependency>
  <groupId>com.demo</groupId>
  <artifactId>order-object</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

现在，在“goods-restapi”模块中创建一个订单消息接收通道“orderschannel”，注意这个通道的名字必须与订单发布通道的名字一样，如下所示：

```
package com.demo.goods.restapi.mqchannel;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.SubscribableChannel;

public interface InputSource {
    String ORDERSCHANNEL = "orderschannel";
    String REPLYCHANNEL = "replychannel";

    @Input("orderschannel")
    SubscribableChannel ordersInput();
}
```



```

@Output("replychannel")
MessageChannel replyOutput();
}

```

其中的“replychannel”通道是用来对消息通道进行回复的，即收到消息之后，回应一个消息，表示消息已经收到。如果没有回应的话，消息通道将会认为订阅者没有收到消息，这样就会在一定时间间隔后重新发布同类消息，一般不会超过 3 个。

使用上面的消息接收通道就可以创建一个消息监听服务来订阅消息生产者的消息，如下所示：

```

@EnableBinding(InputSource.class)
public class OrdersReceive {
    private static Logger logger = LoggerFactory.getLogger(OrdersReceive.class);

    @Autowired
    private GoodsService goodsService;

    @ServiceActivator(inputChannel = InputSource.ORDERSCHANNEL, outputChannel = InputSource.REPLYCHANNEL)
    public CompletableFuture<String> accept(OrderQo orderQo) {
        return CompletableFuture.supplyAsync(() -> {
            if (orderQo != null) {
                logger.info("接收到订单更新消息，订单编号=" + orderQo.getOrderNo());

                if (orderQo.getStatus() != null && orderQo.getStatus() < 0) {
                    List<OrderDetailQo> list = orderQo.getOrderDetails();
                    for (OrderDetailQo orderDetailQo : list) {
                        Goods goods = goodsService.findOne(orderDetailQo.getOrderDetailId());

                        if (goods != null) {
                            Integer num = goods.getBuynum() != null && goods.getBuynum() > 0 ? goods.getBuynum() - 1 : 0;
                            goods.setBuynum(num);
                            goodsService.save(goods);
                            logger.info("更新了商品购买数量，商品名称=" + goods.getName());
                        }
                    }
                }
            }
            return "1";
        });
    }
}

```



```
}
}
```

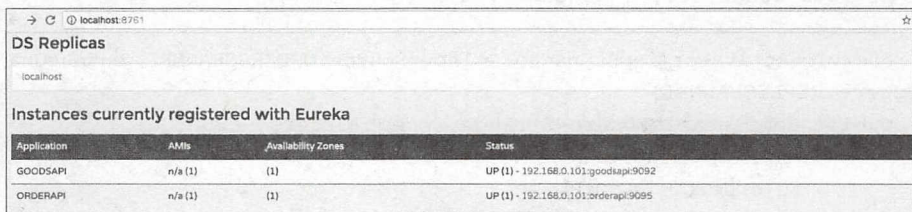
这样，接收到消息之后，在商品服务中就可以根据消息内容进行相关的处理。首先检查订单状态是否为撤销订单，如果订单状态小于零，则对订单明细中的商品购买数进行修改。

7.3.3 使用消息测试

对于上面的设计，我们可以使用一个单元测试来进行验证。

首先，确认已经启动了商品服务的“goodsapi”和订单服务的“orderapi”应用。

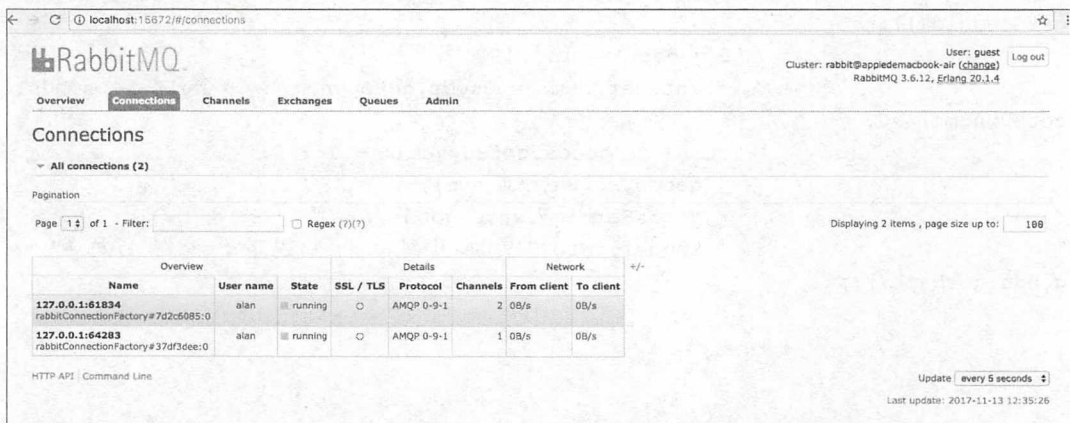
启动之后，通过查看注册管理中心，应该可以看到类似于如图 7-4 所示的情况，表示两个服务已经在注册管理中心进行了注册。



DS Replicas			
localhost			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
GOODSAPI	n/a (1)	(1)	UP (1) - 192.168.0.101:goodsapi:9092
ORDERAPI	n/a (1)	(1)	UP (1) - 192.168.0.101:orderapi:9095

图 7-4 注册中心服务列表

登录 RabbitMQ 服务器，可以查看上面两个服务是否已经连接了服务器，如图 7-5 所示为消息通道的连接情况，可以看到有两个连接。



Overview								Details		Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client				
127.0.0.1:61834 rabbitConnectionFactory#7d2cf085:0	alan	running	○	AMQP 0-9-1	2	0B/s	0B/s				
127.0.0.1:64283 rabbitConnectionFactory#37df3dee:0	alan	running	○	AMQP 0-9-1	1	0B/s	0B/s				

图 7-5 消息通道的连接情况

这样,就可以确认商品服务和订单服务两个服务启动正常,并且已经连接了消息通道。

现在,在订单工程的“order-web”模块中,执行测试程序 `OrderClientTest.java` 中的一个订单修改的测试,即“`update()`”测试用例。这个测试用例将请求“order-restapi”应用服务中的接口来获取一个订单对象,之后,将订单的状态更改为“-1”,然后提交订单修改。测试用例的代码如下所示:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {OrderWebApplication.class})
@SpringBootTest
public class OrderClientTest {
    private static Logger logger = LoggerFactory.getLogger(OrderClientTest.class);

    @Autowired
    private OrderFuture orderFuture;

    @Test
    public void update(){
        String json = orderFuture.findById(2L).join();
        OrderQo orderQo = new Gson().fromJson(json, OrderQo.class);
        orderQo.setStatus(-1);

        String sid = orderFuture.update(orderQo).join();
        logger.info("=====update sid={}", sid);
        assert new Integer(sid) > 0 : "not update";
    }
}
```

测试用例中使用的 `OrderFuture` 是一种非阻塞的异步编程方法的组件服务,将在下一章中再进行详细的介绍。

如果上面的订单和订单明细的商品都存在,并且测试成功通过,将可以在正在运行的订单和商品服务的 `RestAPI` 应用的控制台上看到相关的输出信息。如果在商品服务的 `RestAPI` 应用的控制台上输出了类似如下所示的信息,表明已经成功收到了消息:

```
com.demo.goods.restapi.service.OrdersReceive - 接收到订单更新消息, 订单编号=20170930000001
```

7.4 小结

本章使用领域驱动设计的方法,以实体建模为起点,替代了面向数据结构设计的传统



做法，以一种更高效的设计方法，提高了开发的工作效率，完成了订单领域服务的设计和开发，并结合高性能缓存和异步消息的使用，构建了一个高性能的 **Rest API** 微服务。在整个开发的过程中，通过存储库接口、领域服务和消息通信等单元测试提供了一种验证程序正确性的快速方法，由此可见单元测试在模块开发中的重要作用。

在后续的章节中将以 **Rest API** 所提供的接口服务为基础来进行 **Web UI** 微服务的开发，创建高并发的 **Web** 和 **Wap** 微服务应用。



8

Web UI 微服务开发

Web UI 微服务的开发主要由两大部分来实现：一部分是调用 Rest API 应用的接口以获取数据；另一部分是将获取的数据以合适的视图显示出来。这两部分的内容将由不同的模块来实现，例如，对于订单服务来说，我们用“order-client”模块来实现高并发的接口调用，而“order-web”模块和“order-wap”模块都是实现视图显示的功能模块，它们针对不同终端用户，分别提供了 PC 端和移动设备端的 Web 应用服务。

8.1 高并发接口调用分层设计

Web UI 微服务的开发，首先是接口调用的开发。在接口调用中，我们将集成 FeignClient 接口调用、Hystrix 的断路器功能和 Java 8 的 CompletableFutrue 非阻塞异步编程等各种先进技术，以分层设计的方法实现一种高并发的接口调用的开发，这种调用方法的设计如图 8-1 所示。

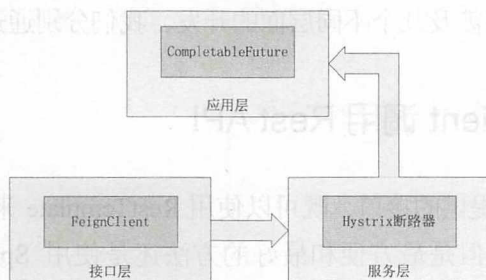


图 8-1 高并发接口调用分层设计

其中，接口层使用 **FeignClient** 调用远程的 **Rest API**，服务层直接调用接口层的方法，并实现 **Hystrix** 断路器的功能，应用层使用非阻塞异步编程技术实现对服务层的高并发调用。

这种接口调用就相当于一个 **Rest API** 微服务的客户端，所以我们使用一个 “client” 模块来管理，这不但是对 **Rest API** 调用的最好封装，而且也可以方便不同应用的引用。

下面以订单服务工程的 “order-client” 模块开发为例来进行说明。

首先，在模块的项目配置中使用如下所示的依赖配置：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
  </dependency>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.0</version>
  </dependency>

  <dependency>
    <groupId>com.demo</groupId>
    <artifactId>order-object</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

在这个配置中，引用了边缘服务组件 **Zuul**，它不但提供了路由代理服务的功能，而且还包含了 **Ribbon** 和 **Hystrix** 等组件的引用。另外，“order-object” 是订单服务的一个查询对象封装。

高并发的接口调用将涉及几个不同层面的开发，我们分别通过以下几个小节来说明。

8.2 通过 FeignClient 调用 Rest API

对 **Rest API** 微服务提供的接口，既可以使用 **RestTemplate** 来调用，也可以使用 **Ajax** 的方式进行访问，但是最方便和最好的方法还是使用 **Spring Cloud** 提供的工具 **FeignClient**，它无论在调用方式，还是参数传递方面，都是非常优秀的。



对于订单服务来说, 针对 Rest API 提供的接口, 可以设计出如下所示的 “OrderClient” 接口:

```
@FeignClient("orderapi")
public interface OrderClient {
    @RequestMapping(method = RequestMethod.GET, value = "/order/{id}")
    String findById(@RequestParam("id") Long id);

    @RequestMapping(method = RequestMethod.GET, value = "/order",
        consumes = MediaType.APPLICATION_JSON_UTF8_VALUE,
        produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    String findPage(@RequestParam("index") Integer index, @RequestParam
("size") Integer size,
        @RequestParam("userid") Long userid, @RequestParam
("merchenatid") Long merchantid,
        @RequestParam("status") Integer status, @RequestParam
("created") String created);

    @RequestMapping(method = RequestMethod.POST, value = "/order",
        consumes = MediaType.APPLICATION_JSON_UTF8_VALUE,
        produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    String create(@RequestBody OrderQo orderQo);

    @RequestMapping(method = RequestMethod.PUT, value = "/order",
        consumes = MediaType.APPLICATION_JSON_UTF8_VALUE,
        produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    String update(@RequestBody OrderQo orderQo);

    @RequestMapping(method = RequestMethod.DELETE, value = "/order/{id}")
    String delete(@RequestParam("id") Long id);
}
```

其中, 通过注解 “@FeignClient” 设定了所引用的 Rest API 应用, 即使用应用的注册实例名称 “orderapi”。使用这一设定, 我们就可以通过实例名称, 而不用通过应用 IP 和端口来访问应用对外暴露的接口。例如, 对于 “/order” 这个接口的调用, 就相当使用了如下所示的链接:

```
http://orderapi/order
```

在 “OrderClient” 的接口声明中, 已经包含在上章中订单服务的 Rest API 设计的所有接口的调用, 即包括获取一个订单信息、订单分页查询、订单生成、订单修改和订单删除等操作。其中, 在参数传递中, 分页查询因为使用了 GET 方法, 它的参数设定使用注解



“@RequestParam”来实现，最终形成的调用方式等同于如下所示的链接方式：

```
http://orderapi/order?index=&size&userid=&merchantid=&status=
```

对于 POST 和 PUT 方法来说，参数传递通过 “@RequestBody” 直接使用查询对象 “OrderQo”。这里需要注意编码格式的设置，我们使用了 “APPLICATION_JSON_UTF8_VALUE”，可以支持中文的编码。

完成 “OrderClient” 的接口设计就实现了对 Rest API 的调用，可以提供给其他应用程序使用。

8.3 使用 Hystrix 断路器

为了使用断路器的功能，我们将设计一个服务层来访问上节设计的接口层，同时在服务层中实现 Hystrix 断路器的功能。

如下所示是订单的一个服务层设计 “OrderRestService” 的实现代码：

```
@Service
public class OrderRestService {
    @Autowired
    private OrderClient orderClient;

    @HystrixCommand(fallbackMethod = "findByIdFallback")
    public String findById(Long id){
        return orderClient.findById(id);
    }

    private String findByIdFallback(Long id){
        OrderQo orderQo = new OrderQo();
        return new Gson().toJson(orderQo);
    }

    @HystrixCommand(fallbackMethod = "findPageFallback")
    public String findPage(OrderQo orderQo){
        String date = null;
        if(orderQo.getCreated() != null) {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            date = sdf.format(orderQo.getCreated());
        }
        return orderClient.findPage(orderQo.getPage(), orderQo.getSize(),
            orderQo.getUserid(),
```




```

        orderQo.getMerchantid(), orderQo.getStatus(), date);
    }

    private String findPageFallback(OrderQo orderQo){
        Map<String, Object> page = new HashMap<>();
        page.put("content", null);
        page.put("totalPages", 0);
        page.put("totalelements", 0);
        return new Gson().toJson(page);
    }

    @HystrixCommand(fallbackMethod = "createFallback")
    public String create(OrderQo orderQo){
        return orderClient.create(orderQo);
    }

    private String createFallback(OrderQo orderQo) {
        return "-1";
    }

    @HystrixCommand(fallbackMethod = "updateFallback")
    public String update(OrderQo orderQo){
        return orderClient.update(orderQo);
    }

    private String updateFallback(OrderQo orderQo) {
        return "-1";
    }

    @HystrixCommand(fallbackMethod = "deleteFallback")
    public String delete(Long id){
        return orderClient.delete(id);
    }

    private String deleteFallback(Long id) {
        return "-1";
    }
}

```

在这个程序设计中，使用注解“@HystrixCommand”来定义断路器的回退方法，当接口服务正常时，则执行正常调用的方法，否则调用回退方法，这样就实现了服务的降级使用机制。在回退方法的设计中，对于一般的查询方法可以返回一个空对象，而需要返回 ID 的地方，可以返回“-1”等参数以提示调用错误，客户端可根据返回参数再作出相应的处理。

8.4 使用非阻塞异步编程方法

对于 node.js 开发者来说，非阻塞异步编程方法是他们引以为傲的地方，幸运的是，Java 8 之后，也能使用这种编程方法了。所谓非阻塞异步编程方法，简单地说，就是一种不用等待返回结果的多线程的回调方法的封装。相比于阻塞式异步回调方法，非阻塞异步编程方法使用一个监听器，这样在使用回调的过程中，就能够自动感知到调用结果，从而实现了高并发的调用方法。

如下所示是使用非阻塞异步编程方法设计的一个订单服务组件：

```
@Component
public class OrderFuture {
    @Autowired
    private OrderRestService ordersService;

    @AsyncTimed
    public CompletableFuture<String> findById(Long id) {
        return CompletableFuture.supplyAsync(() -> {
            return ordersService.findById(id);
        });
    }

    @AsyncTimed
    public CompletableFuture<String> findPage(OrderQo orderQo) {
        return CompletableFuture.supplyAsync(() -> {
            return ordersService.findPage(orderQo);
        });
    }

    @AsyncTimed
    public CompletableFuture<String> create(OrderQo orderQo) {
        return CompletableFuture.supplyAsync(() -> {
            return ordersService.create(orderQo);
        });
    }

    @AsyncTimed
    public CompletableFuture<String> update(OrderQo orderQo) {
        return CompletableFuture.supplyAsync(() -> {
            return ordersService.update(orderQo);
        });
    }
}
```



```

@AsyncTimed
public CompletableFuture<String> delete(Long id) {
    return CompletableFuture.supplyAsync(() -> {
        return ordersService.delete(id);
    });
}
}

```

在这个组件设计中，我们使用了 `CompletableFuture` 的一些函数式调用方法，实现了对服务层的订单服务的高并发调用，那么，`CompletableFuture` 的强势到底是从哪里体现出来的呢？

下面来详细看看 `CompletableFuture` 可以做些什么，它具有哪些强大的功能？

8.4.1 CompletableFuture 介绍

以前，我们使用异步回调的方法可以实现多任务和多线程的处理方式，这无疑是应对高并发访问的一种缓冲方法，但是这种方式有一个致命的弱点那就是阻塞式调用，即使用一个异步回调之后，会有大量的时间耗费在等待之中。

如果我们能在发起一个回调方法之后，使用一个自动监听机制，在回调方法执行完成之后，就主动及时地通知调用者，这样就可以实现非阻塞的回调方法了。像 `node.js`、`Netty` 和 `Google guava` 等开发语言都提供了这种处理方式，这就是一种非阻塞的异步编程方法。

Java 开发语言从 Java 8 之后，也提供了非阻塞的回调处理方法，它封装在 `CompletableFuture` 类中。`CompletableFuture` 提供了非常强大的程序扩展能力，可以帮助我们简化异步编程的复杂性，并通过非阻塞的回调方式处理调用结果。`CompletableFuture` 不但提供了非阻塞异步调用的方法，还可以将处理结果进行转换和结合使用。

如下是一些使用 `CompletableFuture` 进行程序开发的通常使用的方法。

1. 创建 `CompletableFuture` 对象

使用如下所示的 `CompletableFuture` 静态方法，就可以非常方便地创建一个 `CompletableFuture` 对象：

```

public static CompletableFuture<Void> runAsync(Runnable runnable)
public static CompletableFuture<Void> runAsync(Runnable runnable,
Executor executor)
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

```

```
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,
Executor executor)
```

在上面所列的这些方法中，以“Async”结尾并且没有指定 Executor 方法的，将会默认使用 ForkJoinPool.commonPool() 作为它的线程池执行异步代码。

在“OrderFuture”组件设计中，就是使用“supplyAsync”方法来创建 CompletableFuture 对象的。

2. 调用完成的处理方法

调用完成时可以使用如下所示的方法进行处理：

```
CompletableFuture<Void> thenAccept(Consumer<? super T> block);
CompletableFuture<Void> thenRun(Runnable action);
```

3. 输出转换

由于实现了非阻塞的异步回调方法，我们并不需要等待一个调用完成，再通过阻塞或轮询的方式得到结果，而是只要告诉 CompletableFuture，当调用完成的时候执行某个 Function 就可以了，同时还可以进行输出转换。输出转换可以使用如下所示的函数：

```
public CompletableFuture thenApply(Function fn)
public CompletableFuture thenApplyAsync(Function fn)
public CompletableFuture thenApplyAsync(Function fn, Executor executor)
```

这一组函数的功能是在原来的 CompletableFuture 调用完成后，将结果传递给函数“fn”，然后再将“fn”的结果作为新的 CompletableFuture 调用结果，相当于将 CompletableFuture<T> 转换成 CompletableFuture<U>。

这三个函数中不以“Async”结尾的方法，由原来的线程池执行，以“Async”结尾的方法将由默认的线程池 ForkJoinPool.commonPool() 或者指定的线程池 executor 运行。

例如，如下所示为对订单的调用，最后转换成一个视图输出：

```
@RequestMapping(value="/{id}")
public CompletableFuture<ModelAndView> findById(@PathVariable Long id) {
    return orderFuture.findById(id).thenApply(json -> {
        OrderQo orderQo = new Gson().fromJson(json, OrderQo.class);
        return new ModelAndView("order/show", "order", orderQo);
    });
}
```

4. 链接两个调用

如下所示的函数可以将两个调用链接起来一起使用：


```
<U> CompletableFuture<U> thenCompose(Function<? super T,Completable
Future<U>> fn);
```

例如，在商品服务和类目服务两个不同服务中，我们需要使用商品对象的类目 ID 来查询类目对象，可以使用如下所示的方法来编写程序，即将商品查询和类目查询两个调用通过“thenCompose”链接起来：

```
CompletableFuture.supplyAsync(() -> goodsService.findById(id))
    .thenCompose(goods -> CompletableFuture.supplyAsync(() -> {
        GoodsQo goodsQo = new Gson().fromJson(goods, GoodsQo.class);
        String sorts = sortsService.findById(goodsQo.getSortsid());
        return sorts;
    })
    );
```

5. 结合两个转换

如下所示的函数可以将两个转换的结果结合起来，提供给另一个调用使用：

```
<U,V> CompletableFuture<V> thenCombine(CompletableFuture<? extends U> other,
BiFunction<? super T,? super U,? extends V> fn)
```

如下的例子演示了结合两个转换的使用方法，并与链接的使用进行比较：

```
public CompletableFuture<String> getMessage() {
    return CompletableFuture.supplyAsync(() -> {
        return "I am here";
    })
    .thenCombine(CompletableFuture.supplyAsync(() -> " with you"), (ret, msg)
-> ret + msg)
    .thenCompose(p -> CompletableFuture.supplyAsync(() -> p + " and everyone "));
}
```

通过结合转换，再进行链接，最后输出的结果是“I am here with you and everyone”。

6. 多种结合的调用

对两个调用，可以使用链接的方式，对于多个调用还可以使用下列的静态方法：

```
static CompletableFuture<Void> allOf(CompletableFuture<?... cfs)
static CompletableFuture<Object> anyOf(CompletableFuture<?... cfs)
```

最后需要指出的是：不管在创建 `CompletableFuture` 的时候，我们使用了多少层级的组合调用，这些调用并不是嵌套的，而是扁平的，即每一个调用都在非阻塞的情况下自主地进行处理。同时，我们仅仅只需要一步操作，就能获得所有这些组合调用的结果。

8.4.2 性能比较测试

`CompletableFuture` 的非阻塞异步编程的功能确实很强大，它的访问性能表现又怎么样呢？

现在，我们来进行一个性能测试，以比较使用了非阻塞异步编程和没有使用这种编程方法的两种编程方式，在压力测试之中所体现出来的性能指标。

下面在“`order-web`”模块中使用一个简单的测试程序，即使用两种不同风格的编程方式来调用“`order-client`”所提供的接口调用方法。

```
@RestController
@RequestMapping("/test")
public class TestController {
    @Autowired
    private OrderFuture orderFuture;

    @Autowired
    private OrderRestService orderRestService;

    @RequestMapping(value="/list")
    public CompletableFuture<String> findAll() {
        OrderQo orderQo = new OrderQo();
        return orderFuture.findPage(orderQo).thenApply(page -> page);
    }

    @RequestMapping(value="/list1")
    public String findAll1() {
        OrderQo orderQo = new OrderQo();
        return orderRestService.findPage(orderQo);
    }
}
```

其中，第一个链接“`/list`”的设计使用了非阻塞异步编程方法实现，通过调用 `OrderFuture` 来执行分页查询；第二个链接“`/list1`”，没有使用非阻塞异步编程方法，直接调用 `OrderRestService` 实现了相同的功能。

如果将“`order-web`”应用的端口设定为“8095”，即如下的链接为使用非阻塞异步编程方法的调用：

```
http://localhost:8095/test/list
```

而如下的链接为没有使用非阻塞异步编程方法的调用：

`http://localhost:8095/test/list1`

首先,启动“order-restapi”和“order-web”两个应用。

然后,确定上面的链接都能够访问,并都能返回数据。例如,在浏览器中使用上面任何链接,都能返回类似于如图 8-2 所示的结果。

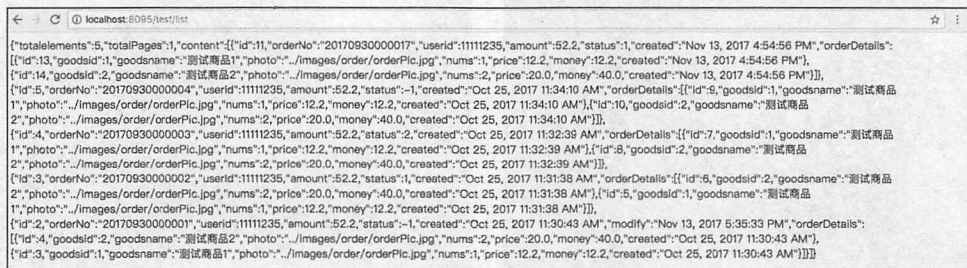


图 8-2 测试链接的调用结果

下面分别使用 Apache 的“ab”和 JMeter 对上面两类链接进行压力测试。

1. “ab”压力测试

Apache 的“ab”是一个简单而又实用的性能测试工具,使用如下两个参数,就可以进行压力测试:

```
-n requests    Number of requests to perform
-c concurrency Number of multiple requests to make
```

其中, -n 表示请求数, -c 表示并发数,例如:

```
ab -n 1000 -c 10 http://localhost:8095/test/list
```

表示使用 1000 个请求、10 个并发的方式对 `http://localhost:8095/test/list` 进行压力测试。

对于这种压力测试,我们只要考虑一个结果“Failed requests: 0”,即当失败请求为零时,表示所有请求已经全部执行,则通过压力测试,否则表示压力测试不能通过。对于不能通过的压力测试,我们再减少并发数进行重新测试,直到所有请求都能通过时,最后确定它的最大可承受的并发数大小。

因为在 Mac OS 中“ab”最大极限只能执行大约 1000 个请求、500 个并发的情况,所以下面的测试将使用 Windows 操作系统来执行,它的最大极限,大约可以执行 1000 个请求、1000 个并发的情况。

经过多次的测试,没有使用非阻塞异步编程方法的调用,在 1000 个请求中,它的最大并发数为 10。类似的执行结果如图 8-3 所示。

即使再增加一个并发数，它就会出现很多失败请求，如图 8-4 所示。

```

C:\Users\Administrator>ab -n 1000 -c 10 http://192.168.0.101:8095/test/list1
This is ApacheBench, Version 2.3 (<Revision: 655654 >)
Copyright 1996 Adan Tuiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.0.101 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:          192.168.0.101
Server Hostname:          8095
Server Port:             
Document Path:            /test/list1
Document Length:          3193 bytes

Concurrency Level:        10
Time taken for tests:      11.846 seconds
Complete requests:         1000
Failed requests:           0
Write errors:              0
Total transferred:         3366400 bytes
HTML transferred:         3193000 bytes
Requests per second:       84.42 [#/sec] (mean)
Time per request:          118.457 [ms] (mean)
Time per request:          11.846 [ms] (mean, across all concurrent requests)
Transfer rate:             277.49 [Kbytes/sec] received

Connection Times (ms)
min  mean[+/-sd] median  max
Connect:    1   10  98.6    4   3116
Processing: 23  108 297.0   77   3237
Waiting:    23   98 294.1   66   3236
Total:      25  117 312.3   83   3239

Percentage of the requests served within a certain time (ms)
50%    83
65%    98
75%   108
80%   114
90%   128
95%   139
98%   153
99%  3190
100% 3239 (longest request)

```

图 8-3 没有使用非阻塞异步编程的最大并发数测试结果

```

C:\Users\Administrator>ab -n 1000 -c 11 http://192.168.0.101:8095/test/list1
This is ApacheBench, Version 2.3 (<Revision: 655654 >)
Copyright 1996 Adan Tuiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.0.101 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:          192.168.0.101
Server Hostname:          8095
Server Port:             
Document Path:            /test/list1
Document Length:          3193 bytes

Concurrency Level:        11
Time taken for tests:      10.847 seconds
Complete requests:         1000
Failed requests:           43
   (Connect: 0, Receive: 0, Length: 43, Exceptions: 0)
Write errors:              0
Total transferred:         3230077 bytes
HTML transferred:         3052163 bytes
Requests per second:       92.19 [#/sec] (mean)
Time per request:          119.313 [ms] (mean)
Time per request:          10.847 [ms] (mean, across all concurrent requests)
Transfer rate:             298.82 [Kbytes/sec] received

Connection Times (ms)
min  mean[+/-sd] median  max
Connect:    1   10  35.5    5   1039
Processing:  3  108 126.6   86  1104
Waiting:    3   88 111.7   71  1104
Total:      5  118 133.5   94  1106

Percentage of the requests served within a certain time (ms)
50%    94
65%   107
75%   119
80%   127
90%   162
95%   218
98%   697
99%  1085
100% 1106 (longest request)

```

图 8-4 出现失败请求的压力测试结果

通过对照测试，使用了非阻塞异步编程的调用方法，在相同请求数的情况下，它可以达到“ab”在单机环境中最大的极限，也就是说，在一个 Windows 操作系统中，1000 个请求加上 1000 个并发的情况下，它还是能全部通过的，如图 8-5 所示。

这说明使用非阻塞异步编程方法的性能比没有使用非阻塞异步编程方法的性能，远远超过了不止 100 倍以上，这种对比是非常惊人的。

如果说简单的“ab”测试工具还不能真正说明真实执行情况的话，下面使用一个比较优秀的压力测试工具，再进行一次性能比较测试。


```

C:\Users\Administrator>ab -n 1000 -c 1000 http://192.168.0.101:8095/test/list
This is ApacheBench, Version 2.3 (<$Revision: 655654 >)
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.0.101 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname:      192.168.0.101
Server Port:          8095

Document Path:        /test/list
Document Length:      3193 bytes

Concurrency Level:    1000
Time taken for tests:  11.762 seconds
Complete requests:    1000
Failed requests:       0
Write errors:         0
Total transferred:    3266000 bytes
HTML transferred:     3193000 bytes
Requests per second:  85.02 [#/sec] (mean)
Time per request:     11761.672 [ms] (mean)
Time per request:     11.762 [ms] (mean, across all concurrent requests)
Transfer rate:        279.48 [Kbytes/sec] received

Connection Times (ms)
      min      mean[+/-std] median      max
Connect:    1       9  22.6         4    3087
Processing: 916 3577 1859.0    2866    9004
Waiting:    914 3550 1853.1    2819    8974
Total:      917 3586 1865.1    2868    9081

Percentage of the requests served within a certain time (ms)
 50%: 2868
 66%: 3724
 75%: 4179
 80%: 4564
 90%: 5148
 95%: 8687
 98%: 8896
 99%: 8968
100%: 9081 (longest request)

```

图 8-5 使用非阻塞异步编程的压力测试结果

2. JMeter 压力测试

使用 JMeter，我们在一个测试计划中设置了 6000 个线程数，并将执行间隔设为 6 秒，分别对上面两种链接执行压力测试。

最终显示的聚合报告如表 8-1 所示，其中，第一条记录为使用了非阻塞异步编程方法的数据；第二条记录为没有使用非阻塞异步编程方法的数据。

表 8-1 JMeter 压力测试聚合报告

Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
6000	39977	41198	53315	60634	63644	3117	66259	17.77%	85.5432	200.46	10.11
6000	4554	5109	8394	9563	10738	3	11472	50.20%	373.925	489.55	22.19

从表 8-1 中并没有看到有特别大的差别，这是因为没有使用非阻塞异步编程方法的测试，很多请求都不能被执行而被丢弃了，所以测试很快就结束了。而使用非阻塞异步编程

方法的测试时间至少比它多了 4 倍以上,所以在聚合报告中,并不能完全体现出它们之间巨大的区别。

如果在测试过程中,同时使用 Hystrix 的监控面板进行观察,就可以更加形象地看到不同的执行情况。

使用非阻塞异步编程方法的运行情况如图 8-6 所示,它出现了很大的访问量,但是断路器还处于关闭的状态,说明系统还处在正常服务的状态。

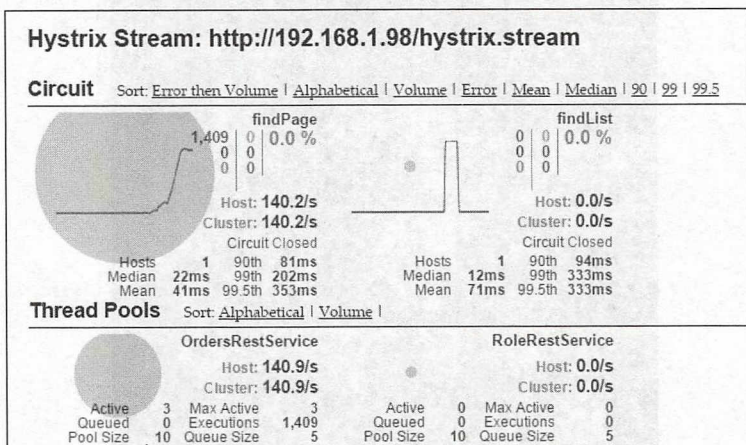


图 8-6 断路器仪表盘正常运行状态

而没有使用非阻塞异步编程方法的运行情况如图 8-7 所示,它的断路器已经处于打开状态,而且故障率在很高的百分比之中,因为很多请求都失败了,所以访问量并不是很大。

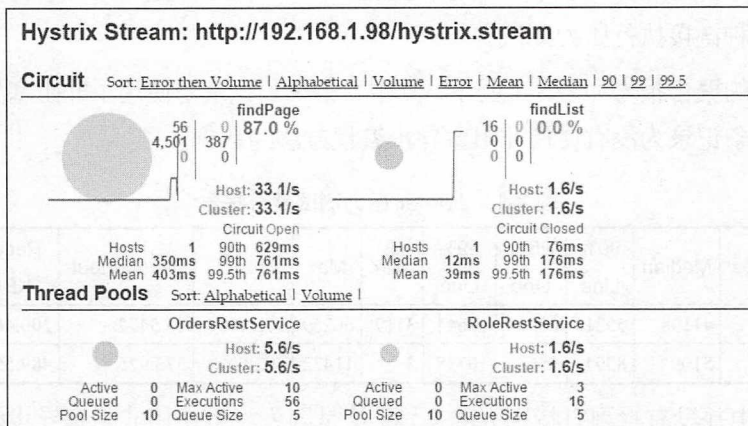


图 8-7 断路器仪表盘故障运行状态

这就可以非常明显地看出,使用非阻塞异步编程方法在很大程度上提高了程序的高并发处理能力,从而可以保证系统始终处于稳定运行状态之中。

8.5 Web 应用开发

完成了高并发的接口调用设计,接下来的任务就是 Web 应用的开发。

Web 应用通过接口调用从 Rest API 中取得数据,然后专注于用户界面的设计,所以这里也可以使用 node.js 或者基于 node.js 的 Angular.js 和 Vue.js 等前端开发工具来设计。但是,从易用性和与 Spring Cloud 默契配合上来看,使用 Spring Boot 框架进行前端开发仍然是一个很不错的选择。

Web UI 微服务的开发根据不同的用户终端,可以分为 PC 端和移动端的两种类型应用,例如,对于订单服务工程来说,分别有“order-web”和“order-wap”两种应用。两种应用的开发基本很类似,只是界面展示上有些不同而已。

下面以订单微服务工程“order-microservice”中的“order-web”模块开发为例来说明 Web 应用的开发方法。

8.5.1 项目引用配置

在“order-web”模块的项目对象模型 pom.xml 设置如下所示的依赖引用:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-commons</artifactId>
        <version>1.11.0.RELEASE</version>
    </dependency>
</dependencies>
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>

<dependency>
  <groupId>com.demo</groupId>
  <artifactId>order-client</artifactId>
  <version>${project.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
</dependencies>
```

这个配置包含如下一些组件的引用：

- Web 组件引用，包括内嵌的 Tomcat 中间件。
- Thymeleaf 组件引用，提供页面设计的模板。
- Feign 组件引用，提供 FeignClient 的支持。
- Eureka 组件引用，提供服务注册和发现的功能。
- “spring-data-commons” 组件引用，提供一些处理数据的对象。
- “order-client” 模块的引用，提供高并发的接口调用方法。

除了一个 Web 应用所必需的这些基本依赖引用之外，我们还可以根据需要增加配置管理中心服务和跟踪服务功能等依赖的引用。

在页面设计中，Spring Boot 使用 Thymeleaf 取代了 JSP，直接使用静态文件的方式即 HTML 来设计用户界面。在高版本的 Spring Boot 中，将不再提供对 JSP 的支持。

这里使用 Thymeleaf 的默认配置，即使用默认的 HTML 页面文件，并将文件存放在“resources/templates”之中。

8.5.2 应用程序配置

首先是 application.yml 配置文件设置，如下所示：

```
server:
  port: 8095
```



```

spring:
  zipkin:
    enabled: true
    base-url: http://localhost:9987

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    preferIpAddress: true

##FeignClient 超时设置
---
spring.cloud.loadbalancer.retry.enabled: true
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:
10000
ribbon.ConnectTimeout: 250
ribbon.ReadTimeout: 1000
ribbon.OkToRetryOnAllOperations: true
ribbon.MaxAutoRetriesNextServer: 2
ribbon.MaxAutoRetries: 1

```

在上面这个配置中，我们设定应用的端口为“8095”，并连接了注册管理中心和服务跟踪设置。而最后的 `FeignClient` 超时设置将针对所有的服务，可以提高客户端访问接口的性能。

然后，在配置文件“bootstrap.yml”中设定了应用的名称为“orderweb”，同时也提供了使用配置管理中心的设置和使用 `RabbitMQ` 消息服务的配置。

```

spring:
  application:
    name: orderweb
  cloud:
    config:
      uri: http://localhost:8888

  rabbitmq:
    addresses: amqp://localhost:5672
    username: alan
    password: alan

```

8.5.3 业务功能开发

在“order-web”模块中，只有一个简单的订单查询的业务功能。通过订单查询的开发，可以体会 Web 应用中业务功能开发的实现方法。

订单查询的设计包含一个首页和一个详情页的设计。其中，在首页中，将以分页的方式来显示订单列表，然后通过订单列表来打开订单详情页。

1. 订单首页设计

首先，使用一个控制器来打开订单的首页视图：

```
@RequestMapping(value="/index")
public ModelAndView index(ModelMap model) throws Exception{
    return new ModelAndView("order/index");
}
```

视图设计是一个页面文件，提供了订单查询的操作和列表显示的布局设计：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" layout:decorator="fragments/layout">
<head>
    <title>订单管理</title>
    <link th:href="@{/scripts/pagination/pagination.css}" rel="stylesheet"
type="text/css" />
    <link th:href="@{/scripts/artDialog/default.css}" rel="stylesheet" type=
"text/css" />
    <link th:href="@{/scripts/My97DatePicker/skin/WdatePicker.css}" rel=
"stylesheet" type="text/css" />
    <script th:src="@{/scripts/pagination/jquery.pagination.js}"></script>
    <script th:src="@{/scripts/jquery.smartselect-1.1.min.js}"></script>
    <script th:src="@{/scripts/artDialog/artDialog.js}" />
    <script th:src="@{/scripts/My97DatePicker/WdatePicker.js}"></script>
    <script th:src="@{/scripts/order/list.js}"></script>
</head>
<body>
<div class="locationLine" layout:fragment="prompt">
    当前位置: 首页 &gt; <em>订单中心</em>
</div>
<div class="statisticBox w-782" layout:fragment="content">
    <form id="queryForm" method="get">
        <div class="radiusGrayBox782">
            <div class="radiusGrayTop782"></div>
            <div class="radiusGrayMid782">
                <div class="dataSearchBox forUserRadius">
```



```

        <ul>
            <li>
                <label class="preInpTxt f-left">用户 ID</label>
                <input type="text" class="inp-list f-left w-200"
value="" id="userid" name="userid"/>
            </li>
            <li>
                <a href="javascript:void(0)" class="blueBtn-62X30
f-right" id="searchBtn">查询</a>
            </li>
        </ul>
    </div>
</div>
</div>
</form>

<div class="dataDetailList mt-12">
    <table class="dataListTab">
        <thead>
            <tr>
                <th>订单号</th>
                <th>金额</th>
                <th>日期</th>
                <th>操作</th>
            </tr>
        </thead>
        <tbody id="tbodyContent">
        </tbody>
    </table>
    <div class="tableFootLine">
        <div class="pagebarList pagination"/>
    </div>
</div>
</div>
</body>
</html>

```

这里，查询操作中使用用户 ID 来作为查询参数，这只是一个简单的实现，除了这个参数之外，还可以使用商家 ID、订单编号、订单状态和订单的生成日期等参数进行查询。

订单列表的显示在视图设计中只是一个布局设计，数据的获取和填充将通过下列的 JS 脚本实现。

其中，获取分页数据使用了如下所示的 JS 脚本：

```

var pageselectCallback = function(page_index, jq, size){
    var html = "" ;
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./list?t='+new Date().getTime(),{
            size:size,page:page_index,userid:${"#userid"}.val()
        },function(data){
            fillData(data.content);
        });
}

```

上面代码中的“./list”链接是一个请求数据的调用，这个调用由一个控制器设计来实现，实现代码如下所示：

```

@RequestMapping(value = "/list")
public CompletableFuture<Page<Map<String, Object>>> findAll(OrderQo
orderQo) {
    return orderFuture.findPage(orderQo).thenApply(json -> {
        Gson gson = TreeMapConvert.getGson();
        TreeMap<String,Object> page = gson.fromJson(json, new TypeToken
<TreeMap<String,Object>>(){}.getType());

        Pageable pageable = new PageRequest(orderQo.getPage(),
orderQo. getSize(), null);
        List<OrderQo> list = new ArrayList<>();

        if(page != null && page.get("content") != null)
            list = gson.fromJson(page.get("content").toString(),
new TypeToken<List<OrderQo>>(){}.getType());
        String count = page.get("totalelements").toString();

        return new PageImpl(list, pageable, new Long(count));
    });
}

```

这里使用 `CompletableFuture` 的非阻塞异步调用方法实现了高并发的调用，从“order-client”组件中获取订单列表数据。其中的数据处理为了方便调用者使用，将 `Json` 结构的数据转换成一个分页对象返回。

回到上面的 JS 脚本设计，在取得数据之后，再使用下列的函数设计将数据逐条填充到视图的列表控件“tbodyContent”之中：


```
function fillData(data){

    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.orderNo == null ? '' : v.orderNo) + '</td>' +
            '<td>' + (v.amount == null ? '' : v.amount.toFixed(2)) + '</td>' +
            '<td>' + (v.created == null ? '' : getSmpFormatDateByLong(v.created,
true)) + '</td>';
        html += '<td><a class="c-50a73f mlr-6" href="javascript:void(0)"
onclick="showDetail(\'\' + v.id + '\')">查看</a>';

        html += '</td></tr>' ;

        $list.append($(html));
    });
}
```

这样，就完成了数据请求和显示的设计。

完成上面设计，最后显示的界面如图 8-8 所示。



图 8-8 订单查询首页设计

2. 订单详情页设计

在首页的列表填充中，我们可以看到如下这行代码：

```
onclick="showDetail(\'\' + v.id + '\')">查看</a>'
```

即在订单列表中，单击订单记录，就可以调用 `showDetail` 函数来显示订单详情。`showDetail` 函数也是用 JS 编写的，如下所示：

```
function showDetail(id){
$.get("./"+id,{ts:new Date().getTime()},function(data){
    art.dialog({
        lock:true,
        opacity:0.3,
        title: "查看信息",
        width:'750px',
        height: 'auto',
        left: '50%',
        top: '50%',
        content:data,
        esc: true,
        init: function(){
            artdialog = this;
        },
        close: function(){
            artdialog = null;
        }
    });
});
}
```

这个设计使用了一个对话框来显示订单记录的详细信息。

其中，对话框设计中所调用的链接由如下所示的控制器实现：

```
@RequestMapping(value="/{id}")
public CompletableFuture<ModelAndView> findById(@PathVariable Long id) {
    return orderFuture.findById(id).thenApply(json -> {
        OrderQo orderQo = new Gson().fromJson(json, OrderQo.class);
        return new ModelAndView("order/show", "order", orderQo);
    });
}
```

这个控制器以一种高并发的调用方法取得一个订单对象的数据，然后将数据通过“order”对象返回给详情页视图。详情页视图的设计如下所示：

```
<div class="addInfBtn">
    <h3 class="itemTit"><span>订单信息</span></h3>
    <table class="addNewInfList">
        <tr>
```



```

        <th>ID</th>
        <td width="240"><input class="inp-list w-200 clear-mr f-left"
type="text" th:value="{order.id}" readonly="true"/></td>
        <th>用户 ID</th>
        <td><input class="inp-list w-200 clear-mr f-left" type="text"
th:value="{order.userid}" readonly="true" /></td>
    </tr>

    <tr>
        <th>明细</th>
        <td>
            <select multiple="multiple" readonly="true">
                <option th:each="details:{order.orderDetails}"
                    th:text="{#strings.length(details.goodsname)>20?#
strings.substring(details.goodsname,0,20)+'...':details.goodsname}"
                    th:selected="true"
                ></option>
            </select>
        </td>
        <th>日期</th>
        <td>
            <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="{order.created} ?
#{dates.format(order.created,'yyyy-MM-dd HH:mm:ss')}" : ''" readonly="true"/>
        </td>
    </tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回
</a>
</div>
</div>

```

在详情页视图设计中，主要使用“th”标签来显示订单对象的数据，并将日期数据进行格式化处理。完成上面设计之后，最后显示的效果如图 8-9 所示。

进行“order-web”应用的调试，必须确认注册中心已经启动。然后再按顺序启动“order-restapi”和“order-web”应用。启动成功后，则可以通过如下链接地址查看效果：

http://localhost:8095/

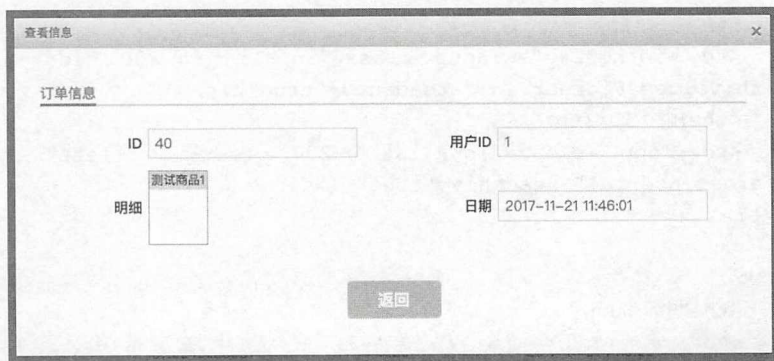


图 8-9 订单详情页

8.6 开发环境的热部署设置

在 Web 应用开发的过程中，我们经常需要对操作界面进行调整，如果需要频繁地重启应用，不但耗时耗力，而且也影响开发的效率和进度。所以，这里顺便说明一下怎么在 IDEA 中进行热部署设置。

首先，在 Web 模块的项目对象模型中增加热部署组件引用和构建工程的插件配置，如下所示：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <fork>true</fork>
      </configuration>
    </plugin>
  </plugins>
</build>
```


然后，在 IDEA 中，通过使用组合键“Shift+Ctrl+Alt+/", 打开“Maintenance”对话框，选择“Registry”，如图 8-10 所示。

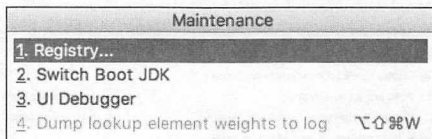


图 8-10 “Maintenance”对话框

在打开的“Registry”配置中，选择“compiler.automake.allow.when.app.running”选项，如图 8-11 所示。

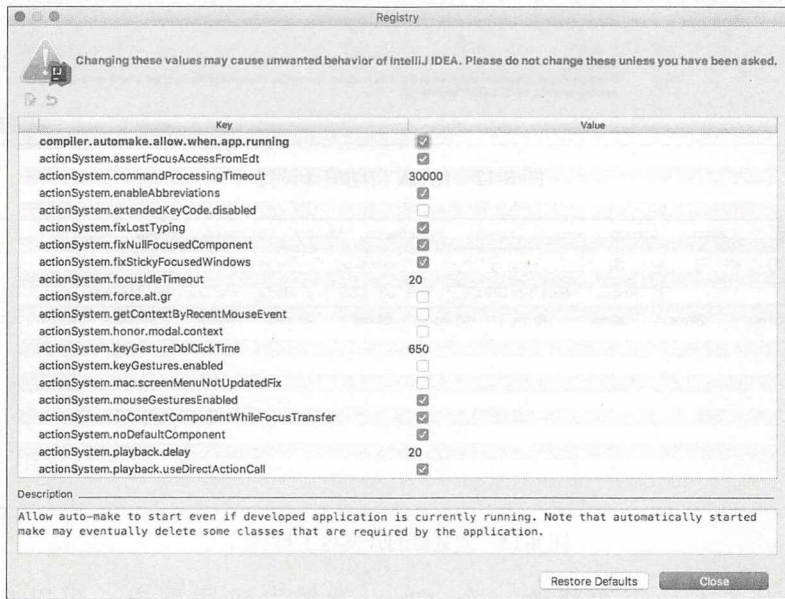


图 8-11 “Registry”配置

上面设置完成之后，针对每一个项目，再在 IDEA 的设置窗口中选择“Compiler”，勾选“Build project automatically”选项，如图 8-12 所示。

最后，再打开浏览器的开发者工具窗口（如 Chrome），选择“Network”菜单下面的“Disable cache”选项，如图 8-13 所示。

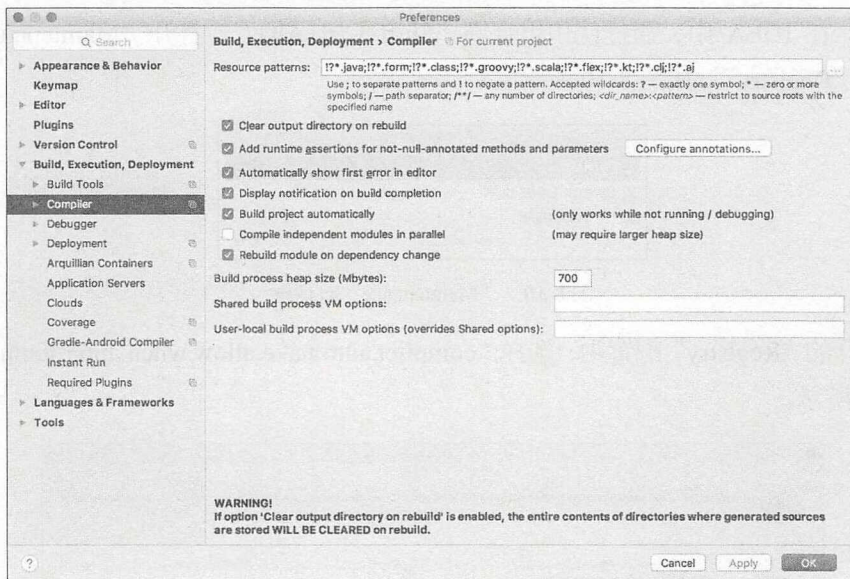


图 8-12 IDEA 自动编译设置

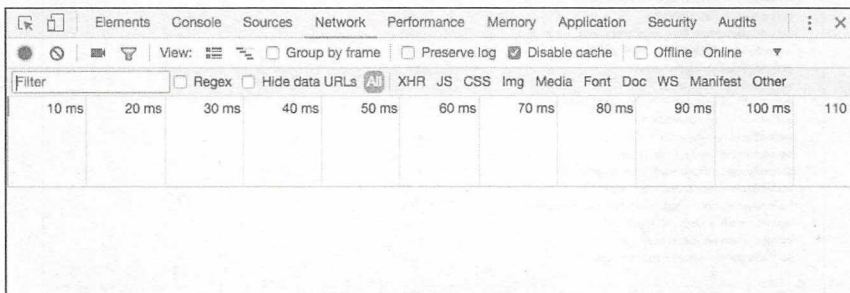


图 8-13 浏览器的开发者工具窗口

另外，为了不让每次修改一个 `Class` 就触发应用重启，可以在配置文件“`application.yml`”中增加如下所示的配置项：

```
spring.devtools.restart.enabled: false
spring.devtools.livereload.enabled: false
```

完成了上面这些配置之后，当我们修改页面设计时，程序就会进行自动更新了。而修改类文件时，还需要手动重启一下，因为大家都不想在每修改一行代码时就自动触发应用重启，那也是很烦人的。

8.7 使用分布式文件系统

微服务应用都将使用 **Docker** 进行部署，并且有可能随时随地部署多个副本，所以必须有一个独立的文件系统来管理用户上传和使用的资源文件，包括图片和视频等。

我们将使用一个轻量级的分布式文件系统（即 **FastDFS**）来提供这种文件管理服务。有关 **FastDFS** 的安装配置可以查看第 14 章的说明。

下面先来介绍怎样在微服务应用中使用分布式文件系统。

在商品服务工程中，用户创建和编辑商品时需要用到图片，这里就以“goods-microservice”项目为例进行说明。

8.7.1 分布式文件系统客户端开发

FastDFS 提供了 Java 语言使用的客户端开发包，但在 **Spring Boot** 框架中使用时还需要进行一些开发，为了简化这种开发过程，我们将使用由 **tobato** 在 **Github** 开源的专为 **Spring Boot** 开发者提供的封装。

首先，在“goods-web”模块中增加如下所示的依赖引用：

```
<dependency>
  <groupId>com.github.tobato</groupId>
  <artifactId>fastdfs-client</artifactId>
  <version>1.25.4-RELEASE</version>
</dependency>
```

接着，在模块的配置文件“application.yml”中增加如下所示的配置：

```
fdfs:
  soTimeout: 1501
  connectTimeout: 601
  thumbImage:
    width: 150
    height: 150
  trackerList:
    - 192.168.1.214:22122
    - 192.168.1.215:22122
  spring.jmx.enabled: false

file.path.head: http://192.168.1.214:8080/
```

这个配置假设 **FastDFS** 的 **TrackerServer** 安装了两台服务器，它们的 IP 地址分别为



“192.168.1.214”和“192.168.1.215”，并且可以通过链接“http://192.168.1.214:8080/”来使用文件。

然后，在工程的启动文件中增加一个如下所示的注解，即导入“fastdfs-client”的配置类：

```
@Import(FdfsClientConfig.class)
```

为了确认上面的引用和配置都已经准备就绪，可以启动一下应用进行验证。如果启动应用正常，就说明上面的一些配置是正确的。

现在，我们就可以创建一个“FastefsClient”来实现文件的上传了，如下所示：

```
@Service
public class FastefsClient {
    @Autowired
    protected FastFileStorageClient storageClient;

    public String uploFile(MultipartFile file){
        String fileType = FilenameUtils.getExtension(file.getOriginalFilename()).toLowerCase();
        StorePath path = null;
        try {
            path = storageClient.uploadFile(file.getInputStream(),
            file.getSize(), fileType, null);
        }catch (IOException e){
            e.printStackTrace();
        }
        if(path != null)
            return path.getFullPath();
        else
            return null;
    }

    public void deleteFile(String fullPath){
        storageClient.deleteFile(fullPath);
    }
}
```

即设计了一个“uploFile”方法通过调用 FastFileStorageClient 实现文件上传，并设计了一个“deleteFile”方法实现文件的删除操作。



8.7.2 商品图片上传设计

在商品编辑中将会用到图片，图片上传的设计通过如下步骤来实现。

首先，设计一个控制器，实现代码如下所示：

```
@RequestMapping("/upload")
public String upload() {
    return "pic/upload-pic";
}

@RequestMapping(value = "/uploadPic", method = RequestMethod.POST)
public void uploadPic(@RequestParam("pictureFile") MultipartFile
multipartFile, HttpServletRequest request, HttpServletResponse response) {
    try {
        String filename = fastefsClient.uploFile(multipartFile);
        Long shopid = 1L;

        AsyncThreadPool.getInstance().execute(new Runnable() {
            @Override
            public void run() {
                try {
                    savePic(multipartFile, filename ,shopid);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

        BufferedImage image = ImageIO.read(multipartFile.getInputStream());

        Map<String, Object> data = new HashMap<String, Object>();
        data.put("pathInfo", pathHead+filename);
        data.put("width", image.getWidth());
        data.put("height", image.getHeight());

        ObjectMapper mapper = new ObjectMapper();
        String ret = mapper.writeValueAsString(data);

        response.setContentType("text/html;charset=utf8");
        response.getOutputStream().write(ret.getBytes());
        response.flushBuffer();
    } catch (IOException e){
        e.printStackTrace();
    }
}
```



这个控制器设计了一个链接“/upload”用来打开上传文件的操作界面，另外，使用另一个链接“/uploadPic”通过调用上节设计的“FastefsClient”实现文件上传，上传后再将图片的路径和文件大小等信息返回给调用者。

文件上传的操作界面主要使用了一个“input”控件实现从操作者的机器上选取文件进行上传，如下所示：

```
<div class="upload-box">
    点击上传
    <input id="pictureFile" name="pictureFile" type="file" class="file"
    onchange="uploadPic_submit(this)"/>
</div>
```

然后，使用 JS 设计一个文件上传的请求方法，可以在视图界面上调用上面的控制器设计中上传文件的链接“/uploadPic”，调用成功后再取出文件信息，实现代码如下所示：

```
function ajaxFileUpload(id){
    var url = '/pic/uploadPic';
    $.ajaxFileUpload({
        url : url, // 需要链接到服务器地址
        fileId : id, // 文件选择框的 id 属性
        dataType : 'json', // 服务器返回的格式，可以是 json
        success : function(data) {
            if(data.errorMsg){
                showMsg(data.errorMsg, "错误");
            }else{
                page.upload.finish(data.pathInfo,data.width,data.height);
            }
        }
    });
}
```

完成上面设计后，最后显示的效果如图 8-14 所示。

其中，在图片上传时，还可以对图片进行裁剪。

8.7.3 富文本编辑器上传文件设计

商品内容的编辑使用了富文本编辑器，可以编辑图文并茂的内容。使用富文本编辑器上



图 8-14 上传图片设计效果

传图片的原理与上小节的图片上传的设计基本是一样的,只是有几点不同的地方需要注意区别对待。这里以使用开源的 **ueditor** 富文本编辑器为例进行说明。

首先,增加一个控制器设计:

```
//ueditor 图片上传
@RequestMapping(value = "/uploadimg", method= RequestMethod.POST,
produces="text/html;charset=UTF-8")
public void uploadimg(@RequestParam("upfile") MultipartFile upfile,
HttpServletRequest request, HttpServletResponse response){
    try {
        String filename = fastefsClient.uploFile(upfile);

        Long shopid = 1L;

        AsyncThreadPool.getInstance().execute(new Runnable() {
            @Override
            public void run() {
                try {
                    savePic(upfile, filename ,shopid);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

        Map<String, Object> data = new HashMap<String, Object>();
        data.put("original", upfile.getOriginalFilename());
        data.put("url", pathHead+filename);
        data.put("title", "");
        data.put("state", "SUCCESS");

        ObjectMapper mapper = new ObjectMapper();
        String ret = mapper.writeValueAsString(data);

        response.setContentType("text/html;charset=utf8");
        response.getOutputStream().write(ret.getBytes());
        response.flushBuffer();
    } catch (Exception e){
        e.printStackTrace();
    }
}
```

这里与上小节的商品图片上传设计不同的地方就是返回参数不一样,主要是根据



ueditor 插件的需要进行设定。

然后, 在新增商品和编辑商品的页面上增加如下所示的代码, 引用 ueditor 插件:

```
<script type="text/javascript" charset="utf-8">
    window.UEEDITOR_HOME_URL = "/ueditor/";
</script>
<script type="text/javascript" charset="utf-8" th:src="@{/ueditor/
editor_config.js}"></script>
<script type="text/javascript" charset="utf-8" th:src="@{/ueditor/
editor_all.js}"></script>
<script type="text/javascript">
    var ME = UE.getEditor('contents',
        {
            wordCount:false,
            initialFrameWidth:406,
            maximumWords:50000,
            wordOverflowMsg:'最多输入 50000 个字符',
            toolbars:[
                ['fullscreen', 'undo', 'redo', '|', 'bold', 'underline',
'strikethrough', 'superscript', 'subscript', 'removeformat',
                'formatmatch', 'autotypeset', '|', 'forecolor',
'backcolor', 'cleardoc', '|', 'rowspacingtop', 'rowspacingbottom',
                'lineheight', '|', 'justifyleft', 'justifycenter',
'justifyright', 'justifyjustify', '|', 'imagenone', 'imageleft',
                'imageright', 'imagecenter', '|', 'customstyle',
'paragraph', 'fontsize', '|', 'emotion', 'date', 'time', 'spechars',
                '|', 'searchreplace', 'insertimage', 'wordimage', 'link']
            ]});
</script>
```

最后, 还必须在 ueditor 插件的配置文件 “editor_config.js” 中更改如下所示的几行配置:

```
//图片上传配置区
, imageUrl: "/pic/uploading" //图片上传提交地址
, imagePath: "" //图片修正地址, 引用了 fixedImagePath, 如有特殊需求, 可自行配置
, imageFieldName: "upfile" //图片数据的 key, 若此处修改, 需要在后台对应文件修改对应参数
```

这样, 就可以实现使用富文本编辑器上传图片文件了。完成设计后使用的效果如图 8-15 所示。



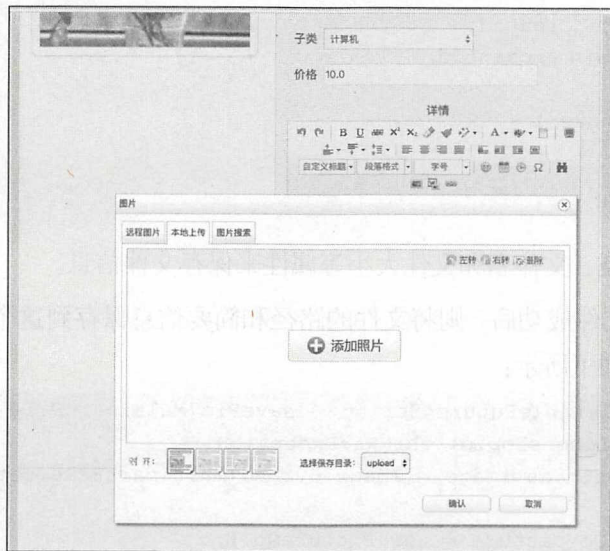


图 8-15 富文本编辑器图片上传的效果

8.7.4 建立本地文件信息库

在一个文件上传之后，为了方便以后可以继续使用这个文件，我们可以在本地建立一个文件信息库，用来保存一个文件的简要信息，实现方法说明如下。

首先，创建一个文件信息库实体，实现代码如下所示：

```
@Entity
@Table(name = "t_picture")
public class Picture {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "path_info")
    private String pathInfo;
    @Column(name = "file_name")
    private String fileName;
    private int width;
    private int height;
    private String flag;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
```

```

    private Date created;
    private Long merchantid;

    public Picture() {
    }
    .....
}

```

即使用文件路径、文件名和文件大小等属性来保存文件信息。

然后，在上传文件成功后，则将文件的路径和简要信息保存到实体之中。文件信息保存的实现代码如下所示：

```

private CompletableFuture<String> savePic(MultipartFile multipartFile,
String filename, Long shopid) throws Exception{
    BufferedImage image = ImageIO.read(multipartFile.getInputStream());

    PictureQo picture = new PictureQo();
    picture.setFileName(filename);
    picture.setHeight(image.getHeight());
    picture.setWidth(image.getWidth());
    picture.setPathInfo(pathHead);
    picture.setMerchantid(shopid);

    return pictureFuture.create(picture).thenApply(sid ->{
        return sid;
    });
}

```

使用了本地文件信息库之后，在创建和编辑商品时就可以使用已经上传的文件。

为了更好地使用本地文件信息库，我们还创建了一个本地文件分页查询的控制器，如下所示：

```

@RequestMapping(value = "/listPic", method = RequestMethod.POST)
@ResponseBody
public CompletableFuture<Page<Map<String, Object>>> listPic(PictureQo
pictureQo){
    return pictureFuture.findPage(pictureQo).thenApply( json -> {
        Gson gson = TreeMapConvert.getGson();
        TreeMap<String,Object> page = gson.fromJson(json, new TypeToken<
TreeMap<String,Object>>(){}.getType());

        Pageable pageable = new PageRequest(pictureQo.getPage(), pictureQo.
getSize(), null);
        java.util.List<GoodsQo> list = new ArrayList<>();
    });
}

```




```

        if(page != null && page.get("content") != null)
            list = gson.fromJson(page.get("content").toString(), new
TypeToken<java.util.List<PictureQo>>(){}.getType());
            String count = page.get("totalelements").toString();
            return new PageImpl(list, pageable, new Long(count));
        });
    }
}

```

控制器使用链接“/listPic”为页面设计提供调用方法,返回本地文件信息的分页列表。

然后,在页面上使用如下所示的 JS 脚本来访问本地文件信息库:

```

function getDataHtml(pageNo,pagesize) {
    $.ajax({
        url: "/pic/listPic",
        dataType: "json",
        type: "POST",
        cache: false,
        data: {page: pageNo-1,size: pagesize || 8},
        success: function (data) {
            var $list = $('#upload-list').empty();
            $.each(data.content, function (i, v) {
                var html = "";
                html += '<div class="upload-item">'+
                    '<div class="img"></div>'+
                    '<p>'+v.width+'x'+ v.height+'</p>'+
                    '<div class="selected"></div>'+
                    '</div>';

                $list.append($(html));
            })
            page.photos.setPosition();
            document.getElementById('pagebar').innerHTML = PageBarNumList.
getPageBar(data.number+1, data.totalPages, 3, 'getDataHtml',pagesize || 8,true);
        },
        error: function (e) {}
    });
}

```

完成设计之后运行效果如图 8-16 所示。



图 8-16 使用本地文件信息库

8.8 小结

本章详细介绍了 Web UI 微服务的开发方法,并使用 Spring Cloud 工具套件中的 Feign 组件、Hystrix 组件和 Java 8 的非阻塞异步编程方法等先进技术,实现了高并发和高可用的设计,然后通过压力测试验证了高并发调用的优异性能。通过 Web 应用的业务开发及其分布式文件系统的使用等,体验了使用 Spring Boot 进行前端开发的便捷性及其与 Spring Cloud 配合使用的优越性。

在后续的章节中将分别介绍电商平台实例中几个应用层面的开发,读者能够更加切实地从中体会高性能的 Rest API 微服务和高并发的 Web UI 微服务所发挥的作用。

9

电商平台移动商城开发

移动商城是电商平台一个非常重要的组成部分，它面向终端用户，为用户提供商品浏览、选购、订单查询和个人信息管理等服务，这些服务分布在不同的应用中，这些应用的设计也分散在各个微服务工程的“wap”模块中。本章介绍怎么将这些分散的应用合并在一起使用，组成一个功能完善的移动商城，体现出微服务架构设计中“合而用之”的概念。

对于一个电商平台来说，移动商城可以说是它的主体部分，它面向广大的用户群体，所以必须具有极高的稳定性，并且能够适应大流量和高并发调用。下面将各个服务工程的移动商城设计部分统一集中在一起进行介绍，以使读者对移动商城的设计有一个全面的认识。

在本书提供的电商平台实例中，移动商城的主要功能包括商品展示、分类查询、购物车、订单查询、个人信息 5 项基本功能。限于篇幅，这里省略了购物车和个人信息方面的设计，其他各个功能的实现在以下几个模块中进行开发：

- 商品微服务工程“goods-microservice”的“goods-wap”模块。
- 类目微服务工程“catalog-microservice”的“catalog-wap”模块。
- 订单微服务工程“order-microservice”的“order-wap”模块。

在这几个模块的设计中都使用了高并发的实现方法，通过调用 Rest API 微服务实现数据管理，而视图方面的设计为适应移动终端设备的特性，使用了 H5 单页设计。

9.1 移动商城首页设计

移动商城首页是商品展示的地方，在“goods-wap”模块中进行开发，这个首页设计包含商品搜索查询和列表显示两大功能。

首先，在控制器设计中提供首页的链接和数据查询功能：

```
@RestController
@RequestMapping("/goods")
public class GoodsController {
    private static Logger logger = LoggerFactory.getLogger(GoodsController.
class);

    @Autowired
    private GoodsFuture goodsFuture;

    @RequestMapping(value="/index")
    public ModelAndView index(ModelMap model) throws Exception{
        return new ModelAndView("goods/index");
    }

    @RequestMapping(value = "/list")
    public CompletableFuture<Page<Map<String, Object>>> findAll(GoodsQo
goodsQo) {
        return goodsFuture.findPage(goodsQo).thenApply(json -> {
            logger.info("goods list = {}", json);
            Gson gson = TreeMapConvert.getGson();
            TreeMap<String, Object> page = gson.fromJson(json, new TypeToken<
TreeMap<String, Object>>(){}.getType());

            Pageable pageable = new PageRequest(goodsQo.getPage(), goodsQo.
getSize(), null);
            List<GoodsQo> list = new ArrayList<>();

            if(page != null && page.get("content") != null)
                list = gson.fromJson(page.get("content").toString(), new
TypeToken<List<GoodsQo>>(){}.getType());
            String count = page.get("totalelements").toString();

            return new PageImpl(list, pageable, new Long(count));
        });
    }
}
```


通过上面的首页链接“/index”，将返回一个 H5 单页设计的页面视图“index.html”。另外，链接“/list”是一个商品列表数据查询设计，使用查询对象 GoodsQo 传递参数，调用 GoodsFuture 的接口封装实现了高并发的数据调用。

页面视图“index.html”的设计，由页面布局和 JS 两部分组成，其中页面布局的实现代码如下所示：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="utf-8" />
    <meta content="yes" name="apple-mobile-web-app-capable"/>
    <meta content="black" name="apple-mobile-web-app-status-bar-style"/>
    <meta name="format-detection" content="telephone=no"/>
    <title>商品查询</title>
    <link th:href="@{/styles/order.css}" rel="stylesheet" type="text/css" />
    <style type="text/css">

    article,aside,dialog,footer,header,section,footer,nav,figure,menu{display
:block}

    </style>
    <script th:src="@{/scripts/jquery-1.10.2.min.js}"></script>
    <script th:src="@{/scripts/viewscale.js}"></script>
    <script th:src="@{/scripts/Pull_Event.js}"></script>
  </head>
  <body>
    <div class="mainBox pt-194">
      <div class="topFixedArea">
        <div class="searchBox">
          <input class="searchInput" id="searchInput" type="text"
placeholder="搜索商品" value="" />
          <label class="searchIcon" for="searchInput"></label>
          <div class="searchGo" ><a href="javascript:searchGo();"
">Go</a></div>
        </div>
        <header class="navigatorBox">
          <nav class="navigator fiveNav">
            <a class="current" href="javascript:void(0)">商品</a>
            <a href="javascript:gotoService('catalogwap', '');">
            <a href="javascript:gotoService('orderwap', '');">
            <a href="javascript:void(0)">购物车</a>
          </nav>
        </header>
      </div>
    </div>
  </body>
</html>
```

```

        <a href="javascript:gotoService('orderwap',
'/order/switch');">账号</a>
    </nav>
</header>
</div>
<section class="orderList">
    <ul class="dataUl">
    </ul>
</section>
<input type="hidden" id="totalPage" value="1"/>
</div>
<div class="copy">关于我们</div>
</body>
</html>

```

在页面的主体设计中主要包含如下几个功能：

- 商品搜索查询。
- 页面导航设计。
- 列表数据显示。

这些功能主要通过 JS 实现，如下所示：

```

<script th:inline="javascript">
    var sortsid = [[${sortsid}]];
</script>
<script>
    /**/
    var goods_name;
    var pageNum = 0;
    $(function () {
        listData(pageNum, 10);
        // 滑动加载更多
        new Pull_Event({
            prompt_selector: '.dataUl',
            prompt_method: 'append', //before|prepend|append|after
            handle: function () {
                var _this = this;
                var totalPage = $('#totalPage').val();
                pageNum++;
                if (pageNum &gt;= totalPage){
                    _this.destroy();
                    return;
                }
            }
        })
        setTimeout(function () {
</pre>
</div>
<div data-bbox="94 922 152 938" data-label="Page-Footer">170</div>
```



```

        listData(pageNum, 10, function () {
            _this.done();
        });
    }, 500);
    }
    });

});

function searchGo(){
    goods_name = $('#searchInput').val();
    //alert(goods_name);
    $(".dataUl").empty();
    listData(pageNum, 10);
}

//刷新页面数据
function listData(pageNum, pageSize, callback)
{
    var $dataUl = $(".dataUl");
    $.ajax({
        url: "./list",
        data: {
            name: goods_name,
            sortsid: sortsid,
            page: pageNum,
            size: pageSize
        },
        type: "GET",
        dataType: "json",
        success: function(data) {
            $('#totalPage').val(data.totalPages);
            var html = '';

            $.each(data.content, function (i, v) {
                html += '<li>';
                html += '<div class="orderInfList">';
                html += '<div class="goodsPicList">';
                html += '<a href="./' + v.id + '"></a>';

                html += '</div>';
                html += '<div class="orderInfTxt clearPb">';
                html += '<p>' + v.name + '</p>';
                html += '<p>价格: ¥' + v.price.toFixed(2) + '</p>';
            });
        }
    });
}

```

```

        html += '<p>已购买: ' + (v.buynum==null?'0':v.buynum)
+ '</p>';

        html += '</div>';
        html += '</div>';
        html += '</li>';

    });
    $dataUl.append(html);
    callback && callback();
}

});
}

function gotoService(name, follow){
    $.get("./service/" + name,{ts:new Date().getTime()},function
(data){
        var url = data;
        if(follow) url += follow;
        window.open(url, "_top");
    });
}
/*]]>*/
</script>

```

这里的分页设计与 PC 端的分页设计有些不同，在 PC 端的设计中，有一个分页的工具条，可以通过单击“下页”和“上页”等按钮进行查询，而这里的分页设计，主要是通过屏幕的滑动下拉事件来完成，当操作界面进行翻页滑动时，将自动完成分页查询。这个功能主要由上面代码中的下拉事件“Pull_Event”实现。

其中，数据的查询和显示由“listData”函数实现，即通过调用链接“./list”来获取数据，然后使用页面上的“”控件输出数据视图。

完成上面的这些设计，就可以进行测试了。

开始测试时，确认注册中心已经启动，然后启动“goods-microservice”工程的“goods-restapi”和“goods-wap”两个应用。启动成功后，使用如下链接进行访问：

<http://localhost:7092>

如果在 PC 端的浏览器上测试，可以打开开发者工具，然后通过选择合适的设备进行查看，如图 9-1 所示。



图 9-1 使用浏览器的开发者工具

如果在手机上测试，确认手机与电脑同处于一个局域网，然后将上面的“localhost”改成电脑上的 IP 地址进行访问。如图 9-2 所示为使用 Android 手机测试的效果。



图 9-2 在 Android 手机上访问商城首页

如图 9-3 所示为使用 iPad 测试的显示效果。



图 9-3 在 iPad 上访问商城首页

从图 9-3 中可以看出，视图的界面将会自动适应屏幕的大小。

9.2 使用负载均衡的导航设计

在商城的首页设计中，我们可以看到一个导航设计，提供了“分类”、“订单”、“购物车”和“个人”等链接的导航。移动商城正是通过这些链接将分散在不同工程中的 Wap 应用合并在一起进行使用的。这里的导航设计不是简单地直接使用 URL 进行访问，而是使用了 Eureka 的服务发现功能，实现了不同应用之间的负载均衡调用。

例如，分类的导航使用了如下所示的设计：

```
<a href="javascript:gotoService('catalogwap', '');">分类</a>
```

这里使用了分类 Wap 应用的实例名称“catalogwap”来访问实例，通过“gotoService”函数进行页面跳转，所以可以在多副本的实例中实现负载均衡调用。其中，“gotoService”函数的设计如下所示：


```
function gotoService(name, follow){
    $.get("./service/" + name, {ts:new Date().getTime()}, function(data) {
        var url = data;
        if(follow) url += follow;
        window.open(url, "_top");
    });
};
```

这样，在操作页面上，在导航栏中单击了“分类”之后，将通过调用“./service/catalogwap”这个链接来取得应用所对应的完整 URL，然后再将操作界面进行跳转。

那么，上面的调用是怎样通过实例名称来取得它的 URL 的呢？这里主要使用了 **DiscoveryClient** 的服务发现功能来实现，如下所示：

```
@Autowired
private DiscoveryClient discoveryClient;

@RequestMapping(value="/service/{name}")
public String getService(@PathVariable String name) {
    List<ServiceInstance> list = discoveryClient.getInstances(name);
    String serviceUri = "./";
    if(list != null && list.size() > 0){
        if(list.size() > 1) {
            Random random = new Random();
            ServiceInstance service = list.get(random.nextInt(list.size()
- 1));

            serviceUri = service.getUri().toString();
        }else {
            ServiceInstance service = list.get(0);
            serviceUri = service.getUri().toString();
        }
    }
    logger.info("serviceUri={}", serviceUri);
    return serviceUri;
}
```

上面代码的实现原理是，首先使用 **DiscoveryClient** 通过服务的名字来查找相关的服务实例列表，然后从实例列表中通过每个注册实例的元数据取得它的完整 URL。所以如果一个实例存在多个副本，就可以实现负载均衡调用。这里的负载均衡算法使用简单的随机算法设计，即从实例的副本列表中随机抓取一个可用的实例。在实际生产应用中，可以根据实际情况来设计顺序轮询、最小使用优先选用等算法。

9.3 按分类查询设计

商城的分类查询设计在工程“catalog-microservice”的“catalog-wap”模块中实现。这里的分类查询只是展示一个一级分类列表，然后通过这个列表所提供的分类 ID 参数再跳转到商品应用中进行商品查询。

首先是分类主页的控制器设计，如下所示：

```
@RestController
@RequestMapping("/sorts")
public class SortsController {
    private static Logger logger = LoggerFactory.getLogger(SortsController.class);

    @Autowired
    private SortsFuture sortsFuture;

    @RequestMapping(value="/index")
    public CompletableFuture<ModelAndView> findAll() {
        return sortsFuture.findList().thenApply(json -> {
            Gson gson = TreeMapConvert.getGson();
            List<SortsQo> sortses = gson.fromJson(json, new TypeToken<List<SortsQo>>(){}.getType());
            logger.info("sorts list json={}", json);
            return new ModelAndView("sorts/index", "sortses", sortses);
        });
    }
}
```

即通过链接“/index”取得分类列表数据，然后返回分类主页视图设计“index.html”。

分类主页视图设计是一个 H5 单页，实现的代码如下所示：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="utf-8" />
        <meta content="yes" name="apple-mobile-web-app-capable"/>
        <meta content="black" name="apple-mobile-web-app-status-bar-style"/>
        <meta name="format-detection" content="telephone=no"/>
        <title>分类查询</title>
        <link th:href="@{/styles/order.css}" rel="stylesheet" type="text/css"/>
        <style type="text/css">
            article,aside,dialog,footer,header,section,footer,nav,figure,
```



```

menu{display:block}
    </style>
    <script th:src="@{/scripts/jquery-1.10.2.min.js}"></script>
    <script th:src="@{/scripts/viewscale.js}"></script>
    <script>
        /**/
        function gotoService(name, follow){
            $.get("./service/" + name,{ts:new Date().getTime()},function
(data){
                var url = data;
                if(follow) url += follow;
                window.open(url, "_top");
            });
        }
        /*]]&gt;*/
    &lt;/script&gt;
&lt;/head&gt;
&lt;body&gt;
    &lt;div class="mainBox pt-194"&gt;
        &lt;div class="topFixedArea"&gt;
            &lt;div class="searchBox"&gt;
                &lt;input class="searchInput" id="searchInput" type="text"
placeholder="搜索商品" value="" /&gt;
                &lt;label class="searchIcon" for="searchInput"&gt;&lt;/label&gt;
            &lt;/div&gt;
            &lt;header class="navigatorBox"&gt;
                &lt;nav class="navigator fiveNav"&gt;
                    &lt;a href="javascript:gotoService('goodswap', '');"&gt;
商品&lt;/a&gt;
                    &lt;a class="current" href="javascript:void(0)"&gt;分类
&lt;/a&gt;
                    &lt;a href="javascript:gotoService('orderwap', '');"&gt;
订单&lt;/a&gt;
                    &lt;a href="javascript:void(0)"&gt;购物车&lt;/a&gt;
                    &lt;a href="javascript:gotoService('orderwap', '/order/
switch');"&gt;账号&lt;/a&gt;
                &lt;/nav&gt;
            &lt;/header&gt;
        &lt;/div&gt;
        &lt;section class="orderList"&gt;
            &lt;ul th:each="sorts:${sortses}"&gt;
                &lt;li
th:onclick="${'javascript:gotoService('goodswap','&quot;'+goods/index?sortsid='+sorts.id+'&quot;');'}"&gt;
</pre>
</div>
<div data-bbox="842 930 897 945" data-label="Page-Footer">177</div>
```

```
</a></p>
    </div>
  </div>
</li>
</ul>
</section>
</div>
<div class="copy">关于我们</div>
</body>
</html>
```

这里，只是简单地使用一个“**th:each**”循环语句将一级分类列表逐条进行显示。当在操作界面上单击一个分类时，将使用分类 ID 作为参数，跳转到商品服务的 Wap 应用中执行商品查询，这种实现相当于执行了如下所示的伪链接：

```
http://goodswap/goods/index?sortsid=sorts.id
```

完成设计之后，分类查询的展示效果如图 9-4 所示。



图 9-4 商城分类查询

9.4 商品详情页设计

商品的详情页设计, 首先通过控制器使用 **GoodsFuture** 实现高并发的数据调用, 然后返回一个页面视图设计 “show.html”, 其中, 控制器的实现代码如下所示:

```
@RequestMapping(value="/{id}")
public CompletableFuture<ModelAndView> findById(@PathVariable Long id) {
    return goodsFuture.findById(id).thenApply(json -> {
        GoodsQo goodsQo = new Gson().fromJson(json, GoodsQo.class);
        return new ModelAndView("goods/show", "goods", goodsQo);
    });
}
```

页面视图设计 “show.html” 是一个 H5 单页, 实现的代码如下所示:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="utf-8" />
        <meta content="yes" name="apple-mobile-web-app-capable"/>
        <meta content="black" name="apple-mobile-web-app-status-bar-style"/>
        <meta name="format-detection" content="telephone=no"/>
        <title>商品内容</title>
        <link th:href="@{/styles/main.css}" rel="stylesheet" type="text/css" />
        <style type="text/css">
            article,aside,dialog,footer,header,section,footer,nav,figure,
            menu{display:block}
        </style>
        <script th:src="@{/scripts/jquery-1.10.2.min.js}"></script>
        <script th:src="@{/scripts/viewscale.js}"></script>
        <script>
            /*<![CDATA[*]
            function gotoService(name, follow){
                $.get("./service/" + name,{ts:new Date().getTime()},function
            (data){
                var url = data;
                if(follow) url += follow;
                window.open(url, "_top");
            });
        }
        /*]]>*/
        </script>
    </head>
    <body>
```

```

<input id="id" name="id" type="hidden" th:value="${goods.id}"/>
<div class="swiper-container" style="height:450px;">
  <div class="swiper-wrapper">
    <div class="swiper-slide">
      
    </div>
  </div>
</div>
<div class="spxq_prize">
  <div class="intro" th:text="${goods.name}">商品名称</div>
  <div class="info">
    <span class="prize">
      <em> 价格: ¥ </em><em id="priceShow1" original="0.1"
th:text="${goods.price}">67</em>
    </span>
  </div>
</div>
<div class="contents">
  <div class="abstract"></div>
  <div th:text="${goods.contents}">
  </div>
</div>
<div class="fix-bottom-buy">
  <input id="goodsid" name="goodsid" type="hidden" th:value="${goods.
id}"/>
  <div class="col-2">
    <a id="addCartBtn" href="javascript:void(0)" onclick="history.
back();" class="btn white">返回商城</a>
    <a id="buyNowBtn" th:href="'javascript:gotoservice(&quot;
orderwap&quot;;, &quot;/order/accounts/'+goods.id+'&quot;;)'" class="btn red">
立即购买</a>
  </div>
</div>
</body>
</html>

```

详情页显示了商品的详细信息，并且提供了“立即购买”的跳转链接。点击“立即购买”按钮后将跳转到订单服务的 Wap 应用中执行相关的操作，相当于使用了如下所示的伪链接：

```
http://orderwap/order/accounts/goods.id
```

完成上面的设计，商品详情页的显示效果如图 9-5 所示。

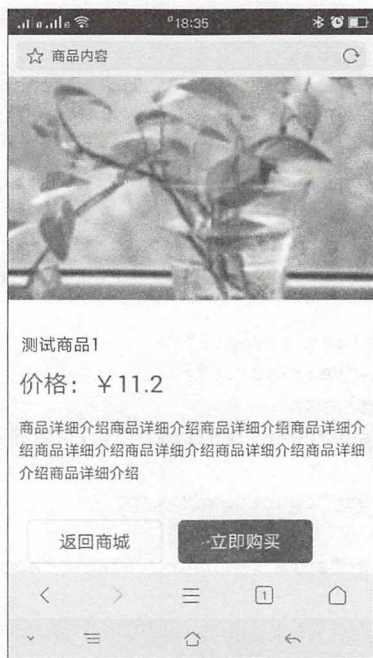


图 9-5 商品详情页

9.5 购买下单实现

购买下单的功能,在订单服务的“order-wap”模块中实现。当从商品详情页中点击“立即购买”按钮后,操作界面将转到订单服务的 Wap 应用中。在订单服务的 Wap 应用中,接收这一操作的控制器设计如下所示:

```
@RequestMapping(value="/accounts/{id}")
public CompletableFuture<ModelAndView> accounts(ModelMap model,
@PathVariable Long id) {
    return goodsFuture.findById(id).thenApply(json -> {
        GoodsQo goodsQo = new Gson().fromJson(json, GoodsQo.class);
        return new ModelAndView("order/accounts", "goods", goodsQo);
    });
}
```

在这里,首先通过商品 ID 取得了商品信息,然后返回一个账号视图设计“accounts.html”。账号视图是一个 H5 单页设计,主要的实现代码如下所示:

```
<body>
    <div class="content prompt1" >
```

```

        <div class="verifyErro">
            <span></span>
            <p class="swit">请登录您的账号! </p>
            <p class="countdown"></p>
        </div>
        <div class="sure"><input class="longinBtn" type="submit" value=
"确定"/></div>
    </div>
    <input id="goodsid" name="goodsid" type="hidden"
th:value="{goods.id}"/>
    <div class="content prompt2" >
        <div class="verifyErro">
            <span></span>
            <p class="swit" th:text="'订单金额: ¥' + ${goods.price}">确
认购买吗? </p>
            <p class="countdown"></p>
        </div>
        <div class="sure"><input class="accountsBtn" type="submit"
value="确定"/></div>
    </div>
    <div class="copy">关于我们</div>
</body>
<script>
    /**/
    $(function(){
        var storage = window.localStorage;
        var user = storage.getItem("user");
        var userid;
        var goodsid = $('#goodsid').val();

        if(user){
            var a = JSON.parse(user);
            userid = a.userid;
            //console.log(a.userid);
            $('.prompt1').hide();
            $('.prompt2').show();
        }else {
            $('.prompt2').hide();
            $('.prompt1').show();
        }

        $('.longinBtn').click(function(){
            window.location.href = "../verify";
        });
    });
</pre>
</div>
<div data-bbox="104 926 162 943" data-label="Page-Footer">182</div>
```



```

$($('.accountsBtn').click(function(){
$.ajax({
url:"../buyone",
data:{
id:goodsid,
userid:userid
},
type: "POST",
dataType: "json",
success:function(data){
if(data && (parseInt(data) > 0)){
alertEC("购买成功! ");
}else{
alertEC("下单失败! ");
}
}
});
setTimeout(function(){
window.location.href="../index";
}, 600);
});
/*]]>*/
</script>

```

在这个设计中,首先对用户账号进行检查,如果用户未登录,即转到登录页面提示用户登录;如果用户已经登录,即提示用户进行购买确认。

如果用户确认购买,即执行购买下单的操作;如果下单成功,即提示“购买成功”,并将操作界面转到订单列表页面。

其中,购买下单的操作使用如下所示的控制器设计:

```

@RequestMapping(value="/buyone", method = RequestMethod.POST)
public CompletableFuture<String> buyone(OrderQo buyone) {
return goodsFuture.findById(buyone.getId()).thenApply(json -> {
GoodsQo goodsQo = new Gson().fromJson(json, GoodsQo.class);
if(goodsQo != null){
Integer sum = 1;
OrderDetailQo orderDetailQo = new OrderDetailQo();
orderDetailQo.setGoodsid(goodsQo.getId());
orderDetailQo.setGoodsname(goodsQo.getName());
orderDetailQo.setPrice(goodsQo.getPrice());
orderDetailQo.setPhoto(goodsQo.getPhoto());
}
});
}

```

```

        orderDetailQo.setNums(sum);
        orderDetailQo.setMoney(sum * goodsQo.getPrice());

        OrderQo orderQo = new OrderQo();
        orderQo.addOrderDetail(orderDetailQo);
        orderQo.setUserid(buyone.getUserid());
        orderQo.setMerchantid(goodsQo.getMerchantid());
        orderQo.setAmount(sum * goodsQo.getPrice());
        orderQo.setOrderNo(new Long((new Date()).getTime()).toString());
        orderQo.setStatus(1); //待发货

        String sid = orderRestService.create(orderQo);

        if(sid != null && new Integer(sid) > 0) {
            Integer buynum = goodsQo.getBuynum() == null ? sum : sum +
goodsQo.getBuynum();
            goodsQo.setBuynum(buynum);
            goodsRestService.update(goodsQo);
            return sid;
        }
        return "-1";
    });
}

```

即根据商品信息和用户信息，调用订单服务创建一个订单。

需要注意的是，在这个购买流程中，我们省略了支付的流程。

9.6 用户登录与账户切换设计

在移动商城的设计中，除了商品和分类查询是完全开发权限的页面，其他涉及个人隐私的个人信息、订单查询和购物车等设计都必须进行权限管理。

用户权限管理的功能在订单服务工程“order-microservice”的“order-wap”模块中开发。这里根据移动设备的特点，使用了本地存储的方式，提供了用户登录和账号切换的设计。

9.6.1 用户登录设计

在用户登录的设计中，为了保证用户身份的真实性，可以由用户提供手机号，并通过短信接收到的验证码进行验证。

使用本地存储设计的用户登录页面是一个 H5 单页设计，主要实现代码如下所示：

```

<body>
  <div class="content">
    <div class="iphone">
      <p class="tips">手机号码</p>
      <input type="text" placeholder="" value="13500001111"/>
    </div>
    <div class="verifyBox">
      <p class="tips">验证码</p>
      <input type="text" placeholder="" value="123456"/>
    </div>
    <div class="verifyError" style="display:none;">
      <span></span>
      <p class="tips">验证码错误</p>
      <p class="countdown"></p>
    </div>
    <div class="sure"><input class="verifyBtn" type="submit" value=
"确定"/></div>
  </div>
  <div class="copy">关于我们</div>
</body>
<script>
/**/
$(function(){
  $('#.verifyBtn').click(function(){
    //验证失败
    //$(".verifyError").show();
    var storage = window.localStorage;

    var customer = new Object();
    customer.phone="13500001111";
    customer.userid="11111235";

    var str = JSON.stringify(customer);
    storage.setItem("user", str);
    window.location.href = "./index";
  });

  $('#.verifyBox').find('input').click(function(){
    $(".verifyError").hide();
  });
});
/*]]&gt;*/
&lt;/script&gt;
</pre>
</div>
<div data-bbox="839 931 894 946" data-label="Page-Footer">185</div>
```

在用户通过验证后，将在本地存储中登记用户的手机号和用户 ID，让用户处于登录状态，直到用户切换账号时，才退出当前登录状态。

需要注意的是，这里的手机号和验证码都是一个模拟数据，同时也省略了从顾客服务中获取用户信息的设计。所以在测试时，直接确认登录后即可保存用户的登录状态。完成设计的运行效果如图 9-6 所示。

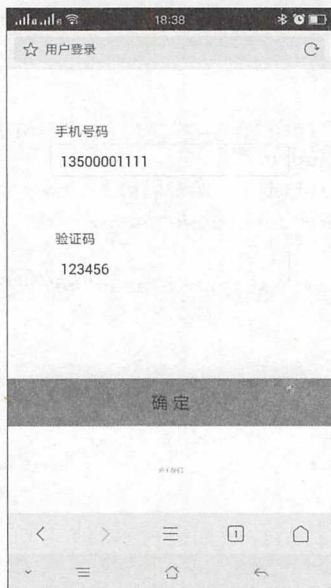


图 9-6 用户登录设计

用户登录之后，在需要身份确认的地方就可以通过本地存储进行处理。

需要注意的是，本地存储只能在一个 Wap 应用中实现，如果在其他应用中需要用到本地存储的用户信息，都必须通过这个应用来获取，这样才能保证用户信息的一致性。

9.6.2 切换账号设计

如果用户没有清除移动设备的缓存，本地存储将长期存在，为了让用户能够退出登录状态或者切换到另一个账号中进行操作，这里提供了一个切换账号的设计。实现这一功能的主要代码如下所示：

```
<body>
  <div class="content">
    <div class="iphone">
```



```

        <p class="swit">切换登录账号</p>
    </div>
    <div class="sure"><input class="switchBtn" type="submit" value=
"确定"/></div>
    </div>
    <div class="copy">关于我们</div>
</body>
<script>
/*<![CDATA[*//
    $(function(){
        $('.switchBtn').click(function(){
            var storage = window.localStorage;
            storage.removeItem("user");
            window.location.href = "./index";
        });
    });
/*]]>*/
</script>

```

在本地存储中通过清除用户登录时保存的用户对象就可以退出登录状态。然后将用户引导到订单查询的主页上，在这里将会自动要求用户进行登录。完成设计的显示效果如图 9-7 所示。

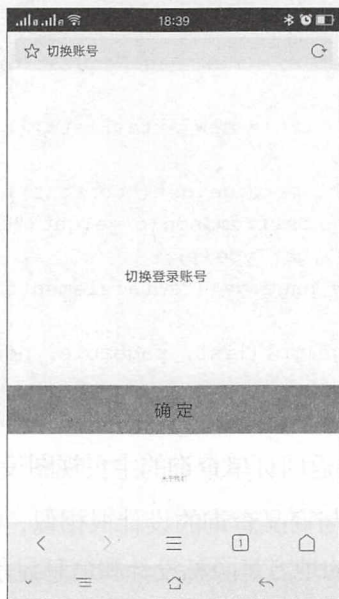


图 9-7 切换账号运行效果

9.7 订单查询设计

在订单服务工程“order-microservice”的“order-wap”模块中还有一个订单查询功能，它是这个应用的主页设计部分，为用户提供订单查询和列表显示的功能。

首先，在控制器设计中提供主页链接和列表数据查询设计：

```
@Autowired
private OrderFuture orderFuture;

@RequestMapping(value="/index")
public ModelAndView index(ModelMap model) throws Exception{
    return new ModelAndView("order/index");
}

@RequestMapping(value = "/list")
public CompletableFuture<Page<Map<String, Object>>> findAll(OrderQo
orderQo) {
    return orderFuture.findPage(orderQo).thenApply(json -> {
        logger.info("order list = {}", json);
        Gson gson = TreeMapConvert.getGson();
        TreeMap<String, Object> page = gson.fromJson(json, new TypeToken<
TreeMap<String, Object>>(){}.getType());

        Pageable pageable = new PageRequest(orderQo.getPage(), orderQo.
getSize(), null);
        List<OrderQo> list = new ArrayList<>();

        if(page != null && page.get("content") != null)
            list = gson.fromJson(page.get("content").toString(), new
TypeToken<List<OrderQo>>(){}.getType());
        String count = page.get("totalelements").toString();

        return new PageImpl(list, pageable, new Long(count));
    });
}
```

其中，通过“/index”链接返回订单查询的主页视图设计“index.html”。

在订单查询视图设计中，与商品查询的设计很相似，同样是通过屏幕的滑动下拉实现了自动分页的功能，而不同的地方是隐私设计和信息显示处理。

为了保证每个用户只能查询自己的订单，在订单列表查询视图的设计中，首先要检

查用户的登录状态, 如果用户未登录, 则提示用户登录, 实现的代码如下所示:

```
<script>
    var storage = window.localStorage;

    var user = storage.getItem("user");
    var userid;
    var orderno;

    if(user){
        var a = JSON.parse(user);
        userid = a.userid;
    }else{
        window.location.href = "./verify";
    }
</script>
```

另外, 在订单的信息处理中, 使用了如下所示的设计:

```
function listData(pageNum, pageSize, callback)
{
    var $dataUl = $(".dataUl");
    $.ajax({
        url: "./list",
        data: {
            orderNo: orderno,
            userid: userid,
            page: pageNum,
            size: pageSize
        },
        type: "GET",
        dataType: "json",
        success: function(data) {
            $('#totalPage').val(data.totalPages);
            var html = '';

            $.each(data.content, function (i, v) {
                html += '<li>';
                html += '<div class="orderInfList">';
                html += '<div class="orderInfTxt clearPb">';
                var orderno = v.orderNo;
                if (parseInt(v.status) == 0) {
                    orderno += "(待付款)";
                } else if (parseInt(v.status) == 1) {
                    orderno += "(待发货)";
                }
            });
        }
    });
}
```

```

    }
    else if (parseInt(v.status) == 2) {
        orderno += " (待收货) ";
    }
    else if (parseInt(v.status) == 3) {
        orderno += " (待评价) ";
    }
    else if (parseInt(v.status) == 4) {
        orderno += " (交易完成) ";
    }
    else if (parseInt(v.status) == -1) {
        orderno += " (已撤销) ";
    }
    else if (parseInt(v.status) == -2) {
        orderno += " (已退款) ";
    }
    }

    html += '<p>订&ensp;单&ensp;号: ' + orderno + '</p>';
    html += '<p>订单金额: ¥' + v.amount.toFixed(2) + '</p>';
    html += '<p>下单时间: ' + new Date(v.created).format
("yyyy-MM-dd HH:mm:ss") + '</p>';
    html += '</div>';
    html += '<div class="orderPicList">';
    $.each(v.orderDetails, function (j, w) {
        html += '';
    });
    html += '</div>';
    html += '</div>';
    html += '</li>';

    });
    $dataUl.append(html);
    callback && callback();
}
});
}

```

即主要对订单状态的显示做了一些转换,并对订单金额的小数位数进行了设定,以避免因为浮点数的原因出现很长的小数位,影响用户体验。

完成设计后的订单列表查询效果如图 9-8 所示。



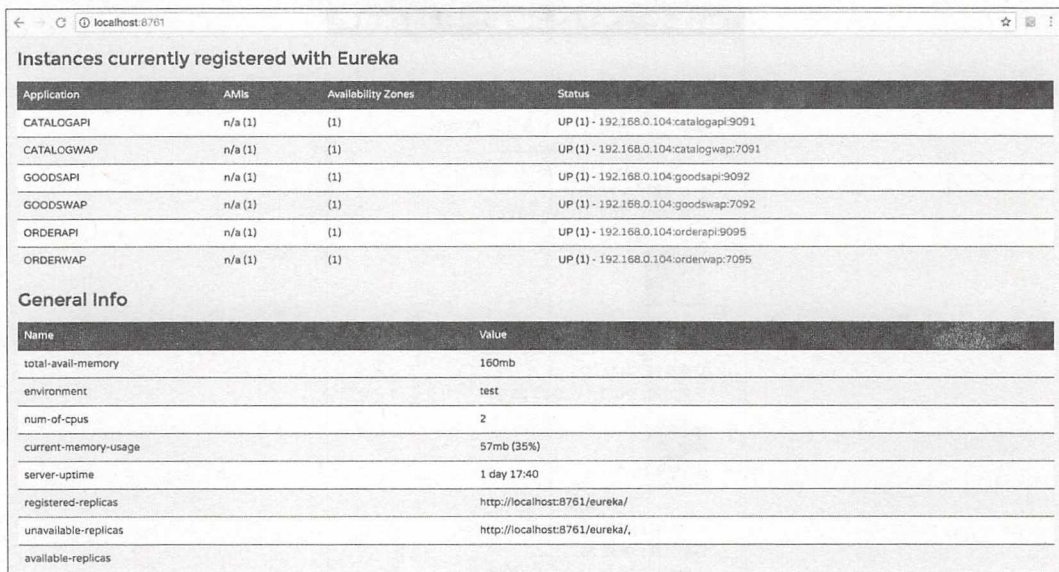
图 9-8 订单列表查询

9.8 集成测试

移动商城设计完成之后可以进行集成测试，在集成测试中必须启动如下应用：

- catalog-restapi。
- catalog-wap。
- goods-restapi。
- goods-wap。
- order-restapi。
- order-wap。

上面的应用都启动完成之后，可以通过注册中心查看所有应用是否启动成功，如图 9-9 所示为全部应用已经启动的情况。



The screenshot shows a web browser window with the address bar set to localhost:8761. The page title is "Instances currently registered with Eureka". Below the title is a table with four columns: Application, AMIs, Availability Zones, and Status. The table lists six applications: CATALOGAPI, CATALOGWAP, GOODSAPI, GOODSWAP, ORDERAPI, and ORDERWAP. Each application has a status of "UP (1)" and a specific IP address. Below the table is a section titled "General Info" which contains another table with two columns: Name and Value. This table lists various system metrics and configuration values.

Application	AMIs	Availability Zones	Status
CATALOGAPI	n/a (1)	(1)	UP (1) - 192.168.0.104:catalogapi:9091
CATALOGWAP	n/a (1)	(1)	UP (1) - 192.168.0.104:catalogwap:7091
GOODSAPI	n/a (1)	(1)	UP (1) - 192.168.0.104:goodsapi:9092
GOODSWAP	n/a (1)	(1)	UP (1) - 192.168.0.104:goodswap:7092
ORDERAPI	n/a (1)	(1)	UP (1) - 192.168.0.104:orderapi:9095
ORDERWAP	n/a (1)	(1)	UP (1) - 192.168.0.104:orderwap:7095

Name	Value
total-avail-memory	160mb
environment	test
num-of-cpus	2
current-memory-usage	57mb (35%)
server-uptime	1 day 17:40
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/
available-replicas	

图 9-9 移动商城测试的应用启动情况

这样就可以使用如下任一链接进行测试：

```
http://localhost:7091
http://localhost:7092
http://localhost:7095
```

测试中，在不同应用之间进行互相跳转时，上面的“localhost”将会自动被 IP 地址所替代。

9.9 小结

移动商城的设计体现了微服务架构设计中“分而治之，合而用之”的原则。移动商城的每一个 Wap 应用都分散在不同的微服务工程中，每个应用都可以进行独立部署、独立运行，并可以根据负载情况进行多副本的发布。而这些分散的应用，我们都可以根据其注册的实例名称通过一种负载均衡调用，组成一个整体进行使用。

下一章将介绍另一种“合而用之”的方法，即通过使用单点登录设计将分布式环境中不同的 Web UI 微服务应用通过统一的用户权限管理，更好地合并在一起进行使用。

商家管理后台与 SSO 设计

在我们所设计的电商平台实例中，商家是这个平台的主角，商家管理后台就是专门为这个主角所提供的安全可靠的操作平台。在商家管理后台中，商家可以进行商品管理、订单管理、物流管理、会员管理、评价管理等各个方面的管理工作。而这些管理及其服务功能的设计，分别由不同的微服务工程所实现，并通过不同应用进行部署。现在我们要做的就是怎么将这些分布在不同应用之中的管理功能，组成一个具有相同的访问控制设计的管理后台。通过使用单点登录设计就可以将这种分散的应用通过统一的权限管理，形成一个有机整体。

单点登录（Single Sign On, SSO）设计可以为分布式环境中的不同应用，提供一个统一的登录认证和授权管理。通过统一的授权认证，商家管理员只要在任何应用之中登录一次，就可以得到使用其他应用的权限。所以，不管商家管理后台的功能由多少个微服务应用组成，这对于一个商家管理员来说，它始终只是一个完整的平台。

商家管理后台的设计和开发主要由商家管理开发和 SSO 开发两部分组成。其中，商家管理主要包含了商家及其权限体系的设计。

这些设计的实现集中在商家微服务项目“merchant-microservice”中进行开发，完整的源代码可以通过如下链接获得：

<https://gitee.com/chenshaojian/merchant-microservice.git>

其中，商家及其权限体系的设计由“merchant-object”、“merchant-domain”、“merchant-restapi”、“merchant-client”、“merchant-web”等模块所组成。单点登录的开发由“merchant-sso”模块和“merchant-security”模块组成。

10.1 商家权限管理体系设计及开发

商家的权限管理体系设计由权限管理模型和菜单管理模型两大功能模型所组成，其中权限管理模型包含商家、用户、角色等实体的设计，菜单管理模型由资源、模块、分类等实体所组成，两大模型之间通过角色与资源的关联关系，组成了一个完整的权限菜单体系结构，如图 10-1 所示。

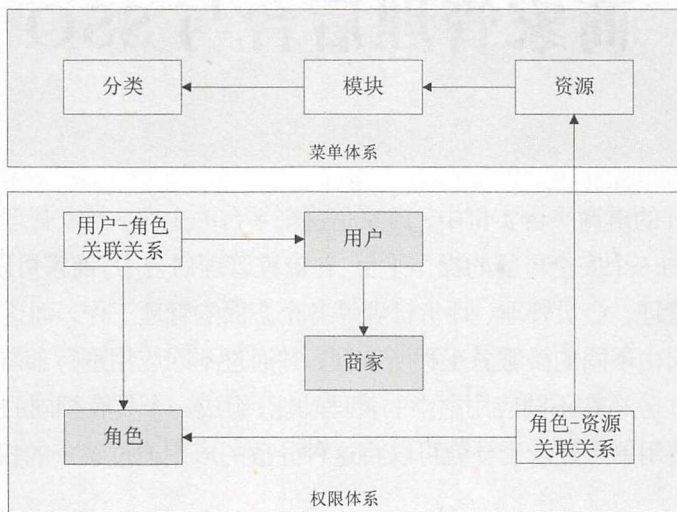


图 10-1 商家权限体系模型结构

在这个模型结构中实体之间的关联关系使用单向关联设计，这些关联关系如下所示：

- 用户从属于商家，是一种多对一的关联关系。
- 用户拥有角色，是一种多对多的关联关系。
- 角色拥有资源，是一种多对多的关联关系
- 资源从属于模块，是一种多对一的关联关系
- 模块从属于分类，是一种多对一的关联关系

在图 10-1 所示的关联关系中，箭头所指一方为关联关系的主键，另一方为外键。其中，用户与角色、角色与资源分别使用了一个中间表来存储关联关系。

10.1.1 商家权限体系建模

商家的权限体系模型主要由商家、用户、角色、资源、模块、分类等实体所组成，下面对这些实体的模型设计分别加以简要说明。

商家实体的模型设计主要由 ID、名称、邮箱、电话、地址、联系人、创建日期等属性所组成，实现代码如下所示：

```
@Entity
@Table(name = "t_merchant")
public class Merchant implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    private String phone;
    private String address;
    private String linkman;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    //    @OneToMany(cascade = { },mappedBy ="merchant")
    //    private List<User> users;

    public Merchant() {
    }
    .....
}
```

注意上面已经注释掉一个关联关系“@OneToMany”（即从商家中关联用户的设计）。因为在后面的用户建模中，将会实现用户与商家的反向关联设计，所以为了避免出现双向关联，这里不再进行关联设计。这样的设计，不但可以提高数据的访问性能，也可以避免出现循环调用的情况。在后面的分类和模块两个实体的设计中，都将遵循这种设计原则。

用户实体模型主要由 ID、名称、密码、邮箱、性别、创建日期等属性所组成，实现代码如下所示：

```
@Entity
@Table(name = "t_user")
```

```

public class User implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String password;
    private String email;
    @Column(name = "sex", length = 1, columnDefinition = "tinyint")
    private Integer sex;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    @ManyToMany(cascade = {}, fetch = FetchType.EAGER)
    @JoinTable(name = "user_role",
        joinColumns = {@JoinColumn(name = "user_id")},
        inverseJoinColumns = {@JoinColumn(name = "role_id")})
    private List<Role> roles;

    @ManyToOne
    @JoinColumn(name = "merchant_id")
    @JsonIgnore
    private Merchant merchant;

    public User() {
    }
    .....
}

```

其中，“@ManyToMany”是一个多对多的正向关联关系，这里使用一个中间表“user_role”来保存关联关系的数据。

“@ManyToOne”是一个反向关联设计，即使用“merchant_id”作为商家对象的外键，与商家实体建立关联关系。

角色实体的模型由 ID、名称、创建日期等属性组成，实现代码如下所示：

```

@Entity
@Table(name = "t_role")
public class Role implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

```



```

    private String name;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default
current_timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    @ManyToMany(cascade = {}, fetch = FetchType.EAGER)
    @JoinTable(name = "role_resource",
        joinColumns = {@JoinColumn(name = "role_id")},
        inverseJoinColumns = {@JoinColumn(name = "resource_id")})
    private List<Resource> resources;

    public Role() {
    }
    .....
}

```

角色与资源也是一个多对多的关联关系，使用“@ManyToMany”进行设置。通过这种关联关系可以将商家权限管理模型与菜单管理模型组成一个完整的权限管理体系。

资源实体模型由 ID、名称、统一资源定位、创建日期等属性所组成，实现代码如下所示：

```

@Entity
@Table(name = "t_resource")
public class Resource implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String url;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    @ManyToOne
    @JoinColumn(name = "mid")
    @JsonManagedReference
    private Model model;

    public Resource() {
    }
}

```

```

    }
    .....
}

```

资源与模块的关联关系同样使用“@ManyToOne”进行反向关联设计，这与上面用户与商家的关联关系设计是一样的道理，主要也是出于性能优化考虑的目的。

模块实体模型由 ID、名称、主机、图标、创建日期等属性所组成，实现代码如下所示：

```

@Entity
@Table(name = "t_model")
public class Model implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String host;
    private String icon;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    @ManyToOne
    @JoinColumn(name = "kid")
    @JsonIgnore
    private Kind kind;

    public Model() {
    }
    .....
}

```

模块的关联关系设计与资源实体的关联设计一样，也是使用“@ManyToOne”进行反向关联设计。

分类实体的模型由 ID、名称、链接、创建日期等属性组成，实现代码如下所示：

```

@Entity
@Table(name = "t_kind")
public class Kind implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)

```



```

private Long id;
private String name;
private String link;
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
@Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
@Temporal(TemporalType.TIMESTAMP)
private Date created;

public Kind() {
}
.....
}

```

分类实体在菜单模型结构中是一个顶级菜单，所以不再需要进行关联设计。

单向关联设计可以提高数据的访问性能，但也有不足的地方。比如，我们在角色建模中，已经实现了角色与资源的单向关联设计，那么，如果从角色中查询资源列表，这是非常容易的，但反过来，要从资源中查询角色列表就有些费力了。为了弥补这种不足，可以使用 SQL 查询语句来实现。

10.1.2 商家权限体系的持久化设计

在商家权限体系的实体建模完成之后，将各个实体与 JPA 的存储库接口进行绑定，就可以为实体赋予操作行为，实现实体的持久化。这一过程其实就是存储库接口设计的工作。

实现商家实体的持久化可以创建一个如下所示的存储库接口：

```

@Repository
public interface MerchantRepository extends JpaRepository<Merchant, Long>,
JpaSpecificationExecutor<Merchant> {

}

```

在这个接口设计中，通过继承 `JpaRepository` 就可以让这个接口具有增、删、改、查的一般操作功能，再通过继承 `JpaSpecificationExecutor` 还可以进行复杂的分页查询设计。如果不用再做其他特殊的查询设计，这样就已经完成了商家实体的持久化设计。

在用户实体存储库接口的设计中，我们还需要做一些扩展功能的实现，使用如下所示的设计：

```

@Repository
public interface UserRepository extends JpaRepository<User, Long>,
JpaSpecificationExecutor<User> {
    @Query("select distinct u from User u where u.name= :name")
    User findByName(@Param("name") String name);

    @Query("select u from User u " +
        "left join u.roles r " +
        "where r.name= :name")
    User findByRoleName(@Param("name") String name);

    @Query("select distinct u from User u where u.id= :id")
    User findById(@Param("id") Long id);

    @Query("select u from User u " +
        "left join u.roles r " +
        "where r.id = :id")
    List<User> findByRoleId(@Param("id") Long id);
}

```

这里多了几个使用注解“@Query”进行自定义查询设计的声明方法。

其中，“findByName”和“findById”主要使用了“distinct”进行去重查询，这可以避免在多对多的关联查询中，出现数据重复的情况。

另外，“findByRoleName”和“findByRoleId”就是上面曾经提到过的，为弥补单向关联的不足而设计的查询，“findByRoleName”实现了从角色名称中查询用户列表的功能，而“findByRoleId”实现了从角色 ID 中查询用户列表的功能。

在角色存储库接口的设计中，也需要增加一个查询设计，如下所示：

```

@Repository
public interface RoleRepository extends JpaRepository<Role, Long>,
JpaSpecificationExecutor<Role> {
    @Query("select o from Role o " +
        "left join o.resources r " +
        "where r.id = :id")
    List<Role> findByResourceId(@Param("id") Long id);
}

```

在这个设计中，“findByResourceId”是一个反向关联的查询，即使用资源 ID 来查询角色列表。

另外几个实体包括分类、模块、资源等的持久化设计，与商家实体的持久化设计类似，只要为其创建一个存储库接口就可以了。

10.1.3 商家权限体系的领域服务开发

领域服务开发是一个服务层的实现，是对存储库接口调用的一种封装设计，这样，可以在存储库接口调用的过程中实现统一的事务管理，以及增加缓存设计等扩展功能。

下面以用户领域服务的开发为例进行说明，其他各个业务模型的领域服务开发与此类似，可以参照这个实例进行开发。

在用户领域服务的设计中，一般增、删、改、查各个操作的设计如下所示：

```
@Service
@Transactional
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private CacheComponent cacheComponent;

    public void save(User user) {
        //删除缓存
        if(CommonUtils.isNotNull(user.getId())){
            String key = user.getId().toString();
            cacheComponent.remove(Constant.MERCHANT_CENTER_USER_ID, key);
//删除原有缓存
        }
        userRepository.save(user);
        //保存缓存
        if(CommonUtils.isNotNull(user.getId())){
            String key = user.getId().toString();
            cacheComponent.put(Constant.MERCHANT_CENTER_USER_ID, key, user,
12); //增加缓存，保存 12 秒
        }
    }

    public void delete(Long id) {
        //删除缓存
        cacheComponent.remove(Constant.MERCHANT_CENTER_USER_ID, id.toString());
        userRepository.delete(id);
    }

    public List<User> findAll() {
        return userRepository.findAll();
    }
}
```

```

    }

    public User findOne(Long id){
        User user = null;
        //使用缓存
        Object object = cacheComponent.get(Constant.MERCHANT_CENTER_USER_ID,
id.toString());
        if (CommonUtils.isNull(object)) {
            user = userRepository.findById(id);
            if (user != null)
                cacheComponent.put(Constant.MERCHANT_CENTER_USER_ID,
id.toString(), user, 12);
        } else {
            user = (User) object;
        }
        return user;
    }

    public List<User> findByRoleId(Long roleId){
        return userRepository.findByRoleId(roleId);
    }

    public User findByName(String name){
        return userRepository.findByName(name);
    }
}

```

在这个设计中，使用注解“@Transactional”实现了隐式的事务管理功能。然后在增、删、改、查的各个操作之中，通过调用 `CacheComponent` 组件增加了使用缓存的功能，在使用缓存的设计中，注意缓存与数据库之间的读写顺序的安排。

用户领域服务的分页查询功能使用了如下所示的设计：

```

public Page<User> findAll(UserQo userQo){
    Sort sort = new Sort(Sort.Direction.DESC, "created");
    Pageable pageable = new PageRequest(userQo.getPage(), userQo.getSize(),
sort);

    return userRepository.findAll(new Specification<User>(){
        @Override
        public Predicate toPredicate(Root<User> root, CriteriaQuery<?>
query, CriteriaBuilder criteriaBuilder) {
            List<Predicate> predicatesList = new ArrayList<Predicate>();

```



```

        if(CommonUtils.isNotNull(userQo.getName())){
            predicatesList.add(criteriaBuilder.like(root.get("name"),
"%"+ userQo.getName() + "%"));
        }
        if(CommonUtils.isNotNull(userQo.getMerchant())){
            predicatesList.add(criteriaBuilder.equal(root.get
("merchant"), userQo.getMerchant().getId()));
        }
        if(CommonUtils.isNotNull(userQo.getCreated())){
            predicatesList.add(criteriaBuilder.greaterThan(root.get
("created"), userQo.getCreated()));
        }

        query.where(predicatesList.toArray(new Predicate[predicates
List.size()]));

        return query.getRestriction();
    }
    }, pageable);
}

```

这里，主要使用了“findAll”方法实现了分页查询的功能，并通过查询对象“userQo”来传递查询参数，这些参数包含用户名称、商家对象和创建日期等属性。

在领域服务的设计中，我们使用了一些查询对象，这些查询对象统一在“merchant-object”模块中实现。查询对象的属性基本上与实体对象的属性互相对应，并且还增加了几个分页查询的属性。

有关查询对象的设计，可以参照用户查询对象的实现，如下所示：

```

public class UserQo extends PageQo implements java.io.Serializable{
    private Long id;
    private String name;
    private String password;
    private String email;
    private Integer sex;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date created;

    private List<RoleQo> roles = new ArrayList<>();

    private MerchantQo merchant;

    public UserQo() {

```

```
}  
.....  
}
```

完成领域服务开发之后，商家权限体系的设计基本上告一段落。下面，我们将使用 Rest API 微服务应用，将商家权限体系组成一个商家服务对外提供的接口调用方法。

10.2 商家管理微服务开发

商家管理微服务是一个独立的 Rest API 应用，这个应用通过接口服务对外提供商家管理、用户权限管理和菜单管理等方面的功能。

商家管理的 Rest API 微服务开发在“merchant-restapi”模块中实现，有关这一类型模块的依赖引用、配置、启动程序的设计等可以参考第 7 章中有关 Rest API 微服务开发的相关说明，这里不再重复。需要注意的是，在这里使用了 Redis 数据库，所以在模块中必须增加如下所示的配置项：

```
spring:  
  redis:  
    host: 127.0.0.1  
    port: 6379  
  # password: 12345678  
  pool:  
    max-idle: 8  
    min-idle: 0  
    max-active: 8  
    max-wait: -1
```

如果还未安装 Redis 数据库，可以参考第 14 章中介绍的方法进行安装。

商家管理微服务将直接调用商家权限体系的领域服务，在调用之前，我们可以对领域服务层进行一个单元测试，以验证领域服务层的程序正确性。

10.2.1 商家领域服务层单元测试

首先，在“merchant-restapi”模块中，可以对上节开发的各个领域服务进行一些测试，通过这一测试，可以对整个商家业务领域的开发进行一个全面的验证。这些测试包括各个实体对象的创建、数据获取及对象更新、删除和分页查询等各个方面的内容。

如下所示是一个创建实体对象的测试用例：




```

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {JpaConfiguration.class, MerchantRestApi
Application.class})
@SpringBootTest
public class UserTest {
    private static Logger logger = LoggerFactory.getLogger(UserTest.class);
    @Autowired
    private UserService userService;
    @Autowired
    private RoleService roleService;
    @Autowired
    private ResourceService resourceService;
    @Autowired
    private ModelService modelService;
    @Autowired
    private KindService kindService;
    @Autowired
    private MerchantService merchantService;

    @Test
    public void insertData(){
        Kind kind = new Kind();
        kind.setName("商家系统");
        kind.setLink("merchantweb");
        kindService.save(kind);
        Assert.assertNotNull(kind.getId(), "create kind error");

        Model model = new Model();
        model.setName("用户管理");
        model.setHost("/user/index");
        model.setKind(kind);
        modelService.save(model);
        Assert.assertNotNull(model.getId(), "create model error");

        Resource resource = new Resource();
        resource.setName("用户修改");
        resource.setUrl("/user/edit/**");
        resource.setModel(model);
        resourceService.save(resource);
        Assert.assertNotNull(resource.getId(), "create resource error");

        Role role = new Role();
        role.setName("商家管理员");
        List<Resource> resources = new ArrayList<>();
    }
}

```



```

resources.add(resource);
role.setResources(resources);
roleService.save(role);
Assert.notNull(role.getId(), "create role error");

Merchant merchant = new Merchant();
merchant.setName("测试商家");
merchantService.save(merchant);
Assert.notNull(merchant.getId(), "create merchant error");

User user = new User();
user.setName("test");
BCryptPasswordEncoder bpe = new BCryptPasswordEncoder();
user.setPassword(bpe.encode("test"));
user.setEmail("user1@com.cn");
List<Role> roles = new ArrayList<>();
roles.add(role);
user.setRoles(roles);
user.setMerchant(merchant);
userService.save(user);
Assert.notNull(user.getId(), "create user error");
}
}

```

在这个测试用例中，已经包含商家业务模型中所有实体对象的创建，这些对象包括分类、模块、资源、角色、商家、用户等实体对象。如果测试通过可以生成一个由分类、模块和资源所组成的三级菜单，同时创建了一个具有所属商家、角色和相关访问资源权限的用户对象，这个用户对象的用户名和密码均为“test”。在后面的开发中，我们可以使用这个用户来登录系统。

如果测试不能通过，可以根据断言中提示的错误信息，在相关的服务组件中查找出错的原因。

获取实体对象的测试，可以使用如下所示的设计：

```

@Test
public void getData(){
    User user = userService.findOne(1L);
    Assert.notNull(user, "not find");
    logger.info("=====name={},role name ={}", user.getName(),
user.getRoles().get(0).getName());
}

```



这个测试用例通过用户 ID 来获取用户信息,测试通过输出用户对象的一些简要信息,包括用户名和用户拥有的第一个角色名称等。

实体对象的更新测试可以使用类似如下所示的设计:

```
@Test
public void update(){
    User user = userService.findByName("user");
    Assert.notNull(user, "user not find");
    Merchant merchant = merchantService.findOne(1L);
    Assert.notNull(merchant, "merchant not find");

    user.setMerchant(merchant);
    userService.save(user);
}
```

这个测试用例首先使用用户名来获取用户对象,然后尝试更改用户的所属商家。如果相关对象都存在的话才能通过这个测试。

删除实体对象的测试可以使用类似如下所示的设计:

```
@Test
public void delData(){
    userService.delete(2L);
}
```

这个测试用例通过用户 ID 来删除一个用户对象。

分页查询的测试可以参照类似于如下所示的设计:

```
@Test
public void findAll() throws Exception{
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    Date date = sdf.parse("2017-01-01 00:00:00");
    UserQo userQo = new UserQo();
    userQo.setCreated(date);

    Merchant merchant = merchantService.findOne(1L);
    MerchantQo merchantQo = CopyUtil.copy(merchant, MerchantQo.class);
    userQo.setMerchant(merchantQo);

    Page<User> page = userService.findAll(userQo);

    Assert.notEmpty(page.getContent(), "list is empty");
    List<User> list = page.getContent();
    for(User user : list){
```



```

        logger.info("=====user name={},role name={},resource
name={},model name={}",
            user.getName(), user.getRoles().get(0).getName(),
            user.getRoles().get(0).getResources().get(0).getName(),
            user.getRoles().get(0).getResources().get(0).getModel().
getName());
    }
}

```

这个测试用例使用查询对象 `UserQo` 配置了分页查询的参数来执行用户信息分页查询。在查询参数中设定了创建日期和所属商家等属性。查询成功后，将输出每条记录的简要信息，这些信息包括用户对象的名称、用户拥有的角色名称、角色关联的资源名称和资源所属的模块名称等基本信息。

如下所示是一个执行分页查询测试的输出信息：

```

com.demo.merchant.restapi.test.UserTest - =====user name=test,role
name=商家管理员,resource name=订单修改,model name=订单管理

```

在进行工程打包时，单元测试还可以作为一个程序正确性的验证手段，如果测试不通过则不能成功打包。

这项功能现在是关闭的，如果要启用这项功能，可以屏蔽掉工程中的下列 `Maven` 配置：

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20</version>
    <configuration>
        <skipTests>true</skipTests>
    </configuration>
</plugin>

```

10.2.2 商家服务的接口开发

在商家管理的 `Rest API` 应用中，包含商家管理、用户权限管理和菜单管理等接口的开发。

每一个接口的设计我们分别使用一个 `RestController` 来实现，这些接口的设计基本上大同小异，下面只以用户接口的设计为例进行说明。

用户的查询接口使用 `GET` 方法实现，如下所示是几种查询接口的实现方法：




```

@RestController
@RequestMapping("/user")
public class UserController {
    private static Logger logger = LoggerFactory.getLogger(UserController.
class);

    @Autowired
    private UserService userService;

    @RequestMapping("/{id}")
    public CompletableFuture<String> findById(@PathVariable Long id) {
        return CompletableFuture.supplyAsync(() -> userService.findOne(id))
            .thenApply(user ->{
                return new Gson().toJson(user);
            });
    }

    @RequestMapping("/names/{name}")
    public CompletableFuture<String> findByName(@PathVariable String name) {
        return CompletableFuture.supplyAsync(() -> userService.findByName
(name))
            .thenApply(user ->{
                return new Gson().toJson(user);
            });
    }

    @RequestMapping("/list")
    public CompletableFuture<String> findList() {
        return CompletableFuture.supplyAsync(() -> userService.findAll())
            .thenApply(users ->{
                return new Gson().toJson(users);
            });
    }

    @RequestMapping(value = "/page")
    public CompletableFuture<String> findPage(Integer index, Integer size,
String name, Long merchantId) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                UserQo userQo = new UserQo();

                if(CommonUtils.isNotNull(index)){
                    userQo.setPage(index);
                }
            }
        });
    }
}

```



```

        if (CommonUtils.isNotNull(size)) {
            userQo.setSize(size);
        }
        if (CommonUtils.isNotNull(name)) {
            userQo.setName(name);
        }
        if (CommonUtils.isNotNull(merchantId)) {
            MerchantQo merchantQo = new MerchantQo();
            merchantQo.setId(merchantId);
            userQo.setMerchant(merchantQo);
        }

        Page<User> users = userService.findAll(userQo);

        Map<String, Object> page = new HashMap<>();
        page.put("content", users.getContent());
        page.put("totalPages", users.getTotalPages());
        page.put("totalelements", users.getTotalElements());

        return new Gson().toJson(page);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
});
}
}

```

这些查询接口包括单个对象查询、列表查询和分页查询等方法的实现，因为是接口调用，查询的结果最终都是以 **Json** 结构的方式返回文本数据的。为了更好地配合客户端高并发的非阻塞异步编程方式的调用，这里同样使用了这种编程方式来实现高并发的调用。

创建用户实体对象使用 **POST** 方法来实现，如下所示：

```

@RequestMapping(value="/save", method = RequestMethod.POST)
public CompletableFuture<String> save(@RequestBody UserQo userQo) throws
Exception{
    return CompletableFuture.supplyAsync(() -> {
        User user = CopyUtil.copy(userQo, User.class);

        List<Role> roleList = CopyUtil.copyList(userQo.getRoles(), Role.
class);
        user.setRoles(roleList);
        user.setMerchant(CopyUtil.copy(userQo.getMerchant(),

```




```
Merchant.class));

        userService.save(user);

        logger.info("新增->ID=" + user.getId());
        return "1";
    });
}
```

创建实体对象时必须将输入参数的查询对象转化为实体对象,使用实体对象调用领域服务进行数据保存。创建用户实体时必须保证用户的合法性,即必须指定用户的所属商家,并为其分配一个角色,这样,这个用户才可以用来登录系统。

实体更新使用 PUT 方法实现,如下所示:

```
@RequestMapping(value="/update", method = RequestMethod.PUT)
public CompletableFuture<String> update(@RequestBody UserQo userQo)
throws Exception{
    return CompletableFuture.supplyAsync() -> {
        User user = CopyUtil.copy(userQo, User.class);

        List<Role> roleList = CopyUtil.copyList(userQo.getRoles(), Role.
class);

        user.setRoles(roleList);
        user.setMerchant(CopyUtil.copy(userQo.getMerchant(), Merchant.
class));

        userService.save(user);

        logger.info("修改->ID=" + user.getId());
        return "1";
    });
}
```

实体的更新设计与实体创建的实现方法很类似,不同的地方就是请求方法不一样。

删除实体对象的设计使用 DELETE 方法实现,如下所示:

```
@RequestMapping(value="/delete/{id}",method = RequestMethod.DELETE)
public CompletableFuture<String> delete(@PathVariable Long id) throws
Exception {
    return CompletableFuture.supplyAsync() -> {
        userService.delete(id);
        logger.info("删除->ID=" + id);
        return "1";
    };
}
```

```

    });
}

```

删除一个实体对象时，有时候还必须要处理好实体的关联关系才能实现删除操作。例如，在角色删除中使用了如下所示的设计：

```

@RequestMapping(value="/delete/{id}",method = RequestMethod.DELETE)
public CompletableFuture<String> delete(@PathVariable Long id) throws
Exception {
    return CompletableFuture.supplyAsync(() -> {
        //让具有此角色的用户脱离关系
        List<User> userList = userService.findByRoleId(id);
        if(userList != null && userList.size() > 0){
            for(User user : userList){
                for(Role role : user.getRoles()){
                    if(role.getId().equals(id)){
                        user.getRoles().remove(role);
                        userService.save(user);
                        break;
                    }
                }
            }
        }
        //安全删除角色
        roleService.delete(id);
        logger.info("删除->ID=" + id);
        return "1";
    });
}

```

即在删除角色之前，要保证角色没有被用户所关联，如果已经存在关联关系，必须将这些关联关系删除之后才能成功删除角色。

完成接口开发之后，可以启动 **Rest API** 应用，对一些查询接口可以使用浏览器进行一个简单的测试。例如，对于用户信息的分页查询可以使用如下所示的链接进行测试：

```
http://localhost:9081/user/page
```

打开链接之后可以看到类似于如图 10-2 所示的 **Json** 结构的数据。



图 10-2 用户接口分页查询测试

这些接口的使用都通过“merchant-client”模块实现高并发的调用封装，有关接口调用的高并发封装的实现方法请参考第 8 章 Web UI 微服务开发中的说明，这里不再赘述。在后面的开发中，我们将使用这些高并发的接口调用封装来调用商家管理的各个接口服务。

10.3 SSO 设计

使用 Spring Cloud Security 和 OAuth2 进行 SSO 设计是非常容易的，这两个组件已经提供了访问控制和授权认证的功能，我们要做的只是使用一些配置和做一些调整就可以实现 SSO 的功能了。

SSO 设计分为服务端和客户端两大部分。SSO 服务端为每个应用提供了统一的访问控制和授权认证服务。SSO 服务端是一个 Web UI 微服务应用，在模块“merchant-ssso”中进行开发，包含用户登录设计、主页设计和认证服务设计等方面的内容。

10.3.1 SSO 基本配置

SSO 的项目管理配置与一般的 Web UI 应用的项目配置基本上相同，可以在 Web UI 应用的项目管理配置的基础上增加 Spring Cloud Security 和 OAuth2 的依赖引用，如下所示：

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-OAuth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-configuration-processor</artifactId>
  <optional>true</optional>
</dependency>

```

其中，**Security** 组件提供了访问控制的功能，**OAuth2** 提供了授权认证的服务。

在应用的配置文件 `application.yml` 中，设定 SSO 应用的服务端口，并设置一个 Cookie 用来保存用户的登录信息，如下所示：

```

server:
  port: 8000
  session:
    cookie:
      name: SESSIONID

```

在配置文件 “`bootstrap.yml`” 中设置应用的名称，如下所示：

```

spring:
  application:
    name: merchantsso

```

如果需要使用配置管理的服务，可以在这个文件中设置相应的调用参数。

10.3.2 在 SSO 中使用商家的权限体系

为了能够使用商家的权限体系进行权限管理，我们需要在 **Spring Security** 的访问控制设计之中使用几个自定义的设计。

首先，通过继承商家的用户对象来创建一个安全用户 **SecurityUser**，并实现 **Spring Security** 的 **UserDetails**，如下所示：

```

public class SecurityUser extends UserQo implements UserDetails
{
  private static final long serialVersionUID = 1L;
  public SecurityUser(UserQo user) {

```



```

        if(user != null)
        {
            this.setId(user.getId());
            this.setName(user.getName());
            this.setEmail(user.getEmail());
            this.setPassword(user.getPassword());
            this.setSex(user.getSex());
            this.setCreated(user.getCreated());
            this.setRoles(user.getRoles());
            this.setMerchant(user.getMerchant());
        }
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Collection<GrantedAuthority> authorities = new ArrayList
<GrantedAuthority>();
        List<RoleQo> roles = this.getRoles();
        if(roles != null)
        {
            for (RoleQo role : roles) {
                SimpleGrantedAuthority authority = new SimpleGrantedAuthority
(role.getName());
                authorities.add(authority);
            }
        }
        return authorities;
    }
    .....
}

```

这里通过查询对象 `UserQo` 来获取商家用户的相关属性,然后通过重写 `UserDetails` 的“`getAuthorities()`”方法来获取用户的操作权限,即用户所拥有的角色列表。

其次,自定义一个 `UserDetailsService`,通过重写“`loadUserByUsername`”取得我们定义的商家用户对象,然后将用户对象复制到上面定义的 `SecurityUser` 之中,实现代码如下所示:

```

@Component
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserFuture userFuture;
}

```

```

@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    String json = userFuture.findByName(username).join();
    UserQo userQo = new Gson().fromJson(json, UserQo.class);
    if (userQo == null) {
        throw new UsernameNotFoundException("UserName " + username + " not found");
    }
    return new SecurityUser(userQo);
}
}

```

最后，通过重写安全管理配置中 `WebSecurityConfigurerAdapter` 的“`configure`”方法，实现了在安全访问控制中使用我们自定义的 `CustomUserDetailsService`，实现代码如下所示：

```

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth.userDetailsService(customUserDetailsService).passwordEncoder(
        passwordEncoder());
}

```

这样，就可以在 `Spring Security` 的访问控制中使用商家用户来登录系统，并进行相应的权限管理。

10.3.3 用户登录设计

现在，我们要提供一个登录界面来接收用户输入用户名和密码等信息，实现登录系统的操作。`Spring Security` 本身有一个登录界面，但是为了与平台的风格互相匹配，我们还是需要自己设计一个。

首先，在安全管理配置 `WebSecurityConfigurerAdapter` 中使用如下所示的设置：

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin().loginPage("/login").permitAll().successHandler(
        loginSuccessHandler())
        .and().authorizeRequests()
        .antMatchers("/images/**", "/checkcode", "/scripts/**", "/styles/**")
        .permitAll()
        .anyRequest().authenticated()
}

```



```

        .and().sessionManagement().sessionCreationPolicy
(SessionCreationPolicy.NEVER)
        .and().exceptionHandling().accessDeniedPage("/deny")
        .and().rememberMe().tokenValiditySeconds(86400).
tokenRepository(tokenRepository()); //记住 24 小时
    }

```

在这个配置中，我们做了如下一些设定：

- 指定登录页面链接：“/login”。
- 指定登录成功的处理程序：“loginSuccessHandler”。
- 忽略对图片等资源的验证。
- 指定拒绝访问的错误提示链接：“/deny”。
- 使用“rememberMe()”设定来记住用户的登录状态。

其次，我们使用一个简单的控制器设计，为链接“/login”返回一个页面设计文件“login.html”，如下所示：

```

@RequestMapping("/login")
public String login(){
    return "login";
}

```

然后，在页面设计文件“login.html”中主要使用一个表单设计，提供了具有用户名和密码等输入控件的操作界面，如下所示：

```

<form th:action="@{/login}" id="loginForm" method="post">
    <ul class="infList">
        <li class="grayBox">
            <label for="username" class="username-icon"></label>
            <input id="username" class="username" name="username" type="text"
placeholder="您的用户名"/>
        </li>
        <li class="grayBox">
            <label class="pwd-icon" id="pwd"></label>
            <input id="password" name="password" class="pwd" type="password"
placeholder="登录密码"/>
        </li>
    </ul>
    <div class="loginBtnBox">
        <input type="button" id="loginBtn" onclick="verSubmit()" value="登
录" class="loginBtn png" />
    </div>
</form>

```

这样就完成了登录设计。登录界面的显示效果类似于一个浮动窗口的样子，如图 10-3 所示。



图 10-3 登录界面设计

10.3.4 有关验证码的说明

当用户第一次登录失败时，将被要求输入验证码。有关验证码的实现，这里主要做两点说明，即验证码的输出和验证的方法。验证码的输出是一个图像，使用如下所示的设计：

```
@RequestMapping(value = "/images/imagecode")
public String imagecode(HttpServletRequest request, HttpServletResponse
response)
    throws Exception {
    OutputStream os = response.getOutputStream();
    Map<String, Object> map = ImageCode.getImageCode(60, 20, os);

    String simpleCaptcha = "simpleCaptcha";
    request.getSession().setAttribute(simpleCaptcha,
map.get("strEnsure").toString().toLowerCase());
    request.getSession().setAttribute("codeTime", new Date().getTime());

    try {
        ImageIO.write((BufferedImage) map.get("image"), "JPEG", os);
    } catch (IOException e) {
        return "";
    }
    return null;
}
```


即在页面上使用“/images/imagecode”这个链接，就可以返回一个包含几个随机数字的验证码图像。在验证码输出的时候，我们同时使用 Session 保存了验证码的数据。验证码的验证实现代码如下所示：

```
@RequestMapping(value = "/checkcode")
@ResponseBody
public String checkcode(HttpServletRequest request, HttpSession session)
    throws Exception {
    String checkCode = request.getParameter("checkCode");
    Object simple = session.getAttribute("simpleCaptcha") ; //验证码对象
    if(simple == null){
        request.setAttribute("errorMsg", "验证码已失效，请重新输入！");
        return "验证码已失效，请重新输入！";
    }

    String captcha = simple.toString();
    Date now = new Date();
    Long codeTime = Long.valueOf(session.getAttribute("codeTime")+"");
    if(StringUtils.isEmpty(checkCode) || captcha == null || !(checkCode.
equalsIgnoreCase(captcha))){
        request.setAttribute("errorMsg", "验证码错误！");
        return "验证码错误！";
    }else if ((now.getTime()-codeTime) / 1000 / 60 > 5){ //验证码有效长度为5分钟
        request.setAttribute("errorMsg", "验证码已失效，请重新输入！");
        return "验证码已失效，请重新输入！";
    }else {
        session.removeAttribute("simpleCaptcha");
        return "1";
    }
}
```

即根据用户的输入，再与上面 Session 保存的数据进行比较，如果数据相同即可确定验证通过。

完成上面的设计，当用户第一次登录失败时，将会显示如图 10-4 所示的登录界面。



图 10-4 需要验证码的登录界面

10.3.5 SSO 的主页设计

如果开始是在一个应用之中进行登录，登录成功之后，SSO 服务端会根据应用的链接地址返回到相关的应用之中；如果开始是在 SSO 服务端进行登录，则默认返回 SSO 服务端的主页，在这个主页中，将会提供访问其他应用的链接。SSO 的主页设计，大体上由两部分组成。

第一部分是一个控制器的设计，实现代码如下所示：

```
@RequestMapping("/")
public CompletableFuture<String> index(ModelMap model, Principal user)
throws Exception{
    return userFuture.findByName(user.getName()).thenApply(json ->{
        UserQo userQo = new Gson().fromJson(json, UserQo.class);
        //分类列表（顶级菜单）
        List<KindQo> kindList = new ArrayList<>();
        List<Long> kindIds = new ArrayList<>();
        for(RoleQo roleQo : userQo.getRoles()){
            for(ResourceQo resourceVo : roleQo.getResources()){
                //去重，获取分类列表
                Long kindId = resourceVo.getModel().getKind().getId();
                if(! kindIds.contains(kindId)){
                    kindList.add(resourceVo.getModel().getKind());
                    kindIds.add(kindId);
                }
            }
        }
    });
}
```



```

    }
  }
}

model.addAttribute("kinds", kindList);
model.addAttribute("user", user);
return "home";
});
}

```

这个设计根据登录用户查询出这个用户能够访问的顶级菜单, 然后根据这个菜单来链接各个应用。其中, **Principal** 对象只提供了一个登录用户的一些简要信息, 我们根据 **Principal** 对象所提供的用户名, 就可以通过商家服务的远程接口查出一个具有完整关联关系的用户对象, 然后围绕用户的关联关系就可以整理出这个用户能够访问的顶级菜单。最后, 使用分类列表 “kinds” 将这个顶级菜单提供给主页视图使用。

第二部分是一个页面视图设计, 其中有关处理分类列表部分的实现代码如下所示:

```

<div class="new-icon" th:each="kind:${kinds}">
  <div class="icon-pic">
    <p><a th:href="'javascript:gotoService('"'+kind.link+'";, &quot;; &quot;);'" class="linka" ></a></p>
  </div>

  <div class="icon-txt">
    <dl>
      <dt>
        <p><a
th:href="'javascript:gotoService('"'+kind.link+'";, &quot;; &quot;);'" class="linka" th:text="${kind.name}"></a></p>
        <span></span>
      </dt>
      <dd>
        <a th:href="'javascript:gotoService('"'+kind.link+
'&quot;;, &quot;; &quot;);'" class="linka" th:text="${kind.name}"></a>
      </dd>
    </dl>
  </div>
</div>

```

这里使用了 Thymeleaf 的一个循环语句 “th:each” 将分类列表中所包含的每一条记录在页面中展示出来。其中, 在顶级菜单链接的设计中使用了 “gotoService” 函数进行页面

跳转，这种跳转将根据菜单中提供的应用实例名称，对相关应用实现负载均衡调用。有关这种跳转设计的详细说明请参考第 9 章中负载均衡导航设计的介绍。完成上面的设计之后，可以进行一个简单的测试。

确认已经启动注册中心，然后启动“merchant-restapi”应用，再启动“merchant-ssso”应用。所有应用启动成功后，在浏览器输入如下所示的链接：

```
http://localhost:8000
```

打开链接后将会进入登录界面。在登录界面上输入上面单元测试中生成的用户名及其密码，即“test/test”。登录成功后，即可打开 SSO 的主页，如图 10-5 所示。

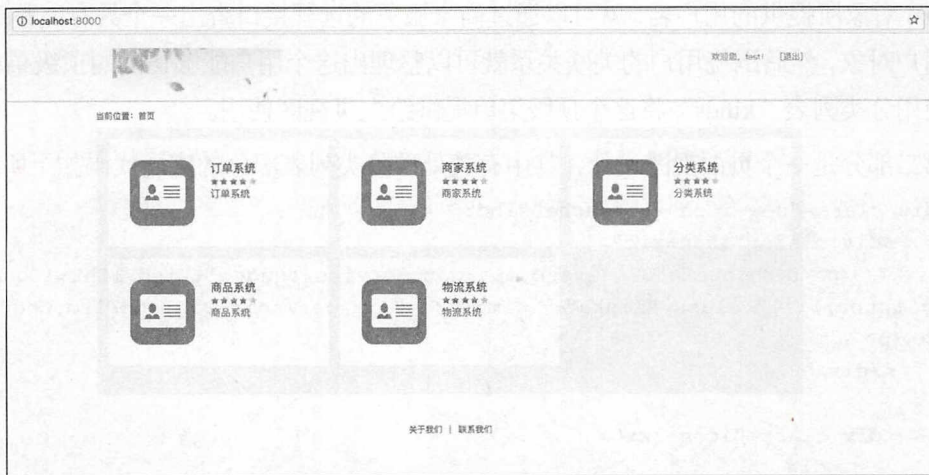


图 10-5 SSO 主页设计

注意：图中所显示的应用列表，将由用户所拥有的权限决定。

10.3.6 OAuth2 服务端设计

OAuth2 服务端可以为使用 SSO 的应用提供授权认证的服务。当用户在一个应用登录之后，就可以通过免登录的认证方式授权使用其他应用。

OAuth2 服务端设计主要通过一个配置类来完成，实现的代码如下所示：

```
@Configuration
@EnableAuthorizationServer
public class OAuthConfigurer extends AuthorizationServerConfigurerAdapter {

    @Bean
```



```

    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        KeyPair keyPair = new KeyStoreKeyFactory(new ClassPathResource(
            "keystore.jks"), "tc123456".toCharArray()).getKeyPair
("tycoonclient");
        converter.setKeyPair(keyPair);
        return converter;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory().withClient("ssoclient").secret("ssosecret")
            .autoApprove(true)
            .authorizedGrantTypes("authorization_code", "refresh_token").
scopes("openid");
    }

    @Override
    public void configure(AuthorizationServerSecurityConfigurer security)
        throws Exception {
        security.tokenKeyAccess("permitAll()").checkTokenAccess(
            "isAuthenticated()").allowFormAuthenticationForClients();
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
        throws Exception {
        endpoints.accessTokenConverter(jwtAccessTokenConverter());
    }
}

```

其中，使用注解“@EnableAuthorizationServer”启用了认证服务器的功能。这里使用令牌（token）的鉴权机制为 SSO 客户端提供授权认证的方法。

管理令牌 JWT（Json Web Token）需要创建一个数字证书，可以使用 Java 的安装程序中的 keystore 工具来生成。项目工程中已经生成了一个数字证书，保存在“keystore.jks”文件中。如果要重新生成数字证书，可以按生成时输入的密码和别名更改上面相关的配置。

这里为 SSO 客户端的凭证配置设定了客户端 ID 和密钥，分别是“ssoclient”和“ssosecret”，这两个参数将用于接入 SSO 的客户端配置中。

另外,为了省略认证的环节,这里在客户端的认证配置中增加了一个“autoApprove”的自动授权确认的配置,这样,在不同应用中进行切换时就不需要用户再进行授权确认。

10.4 SSO 客户端设计

我们将 SSO 客户端设计的实现通过模块“merchant-security”进行了封装,这样可以方便各个接入 SSO 的应用进行调用。

10.4.1 客户端的项目管理配置

SSO 客户端的项目管理使用了如下所示的依赖配置:

```
<dependencies>
  <dependency>
    <groupId>com.demo</groupId>
    <artifactId>merchant-client</artifactId>
    <version>${project.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-OAuth2</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
</dependencies>
```

这个配置除了主要引用的“security”和“OAuth2”组件用来实现应用的安全管理和

授权认证功能之外，还引用了“merchant-client”以提供调用商家服务接口的功能，引用了“redis”以提供使用缓存的功能。

10.4.2 客户端的安全管理配置

在客户端要启用 Spring Security 的安全管理功能和 OAuth2 的 SSO 客户端的功能，主要是通过如下所示一个配置程序来实现的：

```
@Configuration
@EnableOAuth2Sso
@EnableConfigurationProperties(SecuritySettings.class)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private SecuritySettings settings;

    @Autowired
    private RoleFuture roleFuture;
    @Autowired
    private CacheComponent cacheComponent;

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
            .antMatcher("/*").authorizeRequests()
            .antMatchers(settings.getPermitall().split(",")).permitAll()
            .anyRequest().authenticated()
            .and().csrf().requireCsrfProtectionMatcher(csrfSecurityReque
stMatcher())

            .csrfTokenRepository(csrfTokenRepository()).and()
            .addFilterAfter(csrfHeaderFilter(), CsrfFilter.class)
            .logout().logoutUrl("/logout").permitAll()
            .logoutSuccessUrl(settings.getLogoutsuccssurl())
            .and()
            .exceptionHandling().accessDeniedPage(settings.getDeniedpage());
    }

    @Bean
    public CustomFilterSecurityInterceptor customFilter() throws Exception{
        CustomFilterSecurityInterceptor customFilter = new CustomFilter
SecurityInterceptor();
        customFilter.setSecurityMetadataSource(securityMetadataSource());
    }
}
```

```

        customFilter.setAccessDecisionManager(accessDecisionManager());
        customFilter.setAuthenticationManager(authenticationManager);
        return customFilter;
    }

    @Bean
    public CustomAccessDecisionManager accessDecisionManager() {
        return new CustomAccessDecisionManager();
    }

    @Bean
    public CustomSecurityMetadataSource securityMetadataSource() {
        return new CustomSecurityMetadataSource(roleFuture, cacheComponent);
    }
}

```

这个配置程序主要实现了如下一些功能：

- 使用注解“@EnableOAuth2Sso”启用了应用的 SSO 客户端的功能。
- 通过重写 WebSecurityConfigurerAdapter 的“configure”方法来使用一些自定义的安全配置。其中 SecuritySettings 是一个自定义的配置类，它提供了成功退出时的链接配置、允许访问的链接配置和拒绝访问的链接配置等设置参数。
- 在配置中通过一个自定义过滤器“customFilter”引入了其他几个以“custom”开头的自定义设计，其中包括安全管理元数据和权限管理等设计。

10.4.3 权限验证实现原理

用户权限验证主要由两部分设计所实现，分别为安全资源元数据管理和用户访问资源权限检查设计。

安全资源元数据管理的设计的实现代码如下所示：

```

@Override
public Collection<ConfigAttribute> getAttributes(Object object)
    throws IllegalArgumentException {
    String url = ((FilterInvocation) object).getRequestUrl();

    //先从缓存中读取角色列表
    Object objects = cacheComponent.get(MERCHANT_CENTER_ROLES_ALL, "LIST");
    List<RoleQo> roleQoList = null;
    if (CommonUtils.isNull(objects)) {
        roleQoList = loadResourceWithRoles().join();
    }
}

```



```

//如果缓存不存在，从 API 中读取角色列表
    } else{
        roleQoList = ( List<RoleQo>)objects;
    }

    Collection<ConfigAttribute> roles = new ArrayList<>();//有权限的角色列表

    //检查每个角色的资源，如果与请求资源匹配，则加入角色列表，为后面权限检查提供依据
    if(roleQoList != null && roleQoList.size() > 0) {
        for (RoleQo roleQo : roleQoList) { //循环角色列表
            List<ResourceQo> resourceQos = roleQo.getResources();
            if(resourceQos != null && resourceQos.size() > 0) {
                for (ResourceQo resourceQo : resourceQos) { //循环资源列表
                    if (resourceQo.getUrl() != null && pathMatcher.match
(resourceQo.getUrl()+"/*", url)) {
                        ConfigAttribute attribute = new SecurityConfig
(roleQo.getName());

                        roles.add(attribute);
                        //logger.info("加入权限角色列表==角色资源: {}, 角色名称:
{}===", resourceVo.getUrl(), roleVo.getName());
                        break;
                    }
                }
            }
        }
    }

    return roles;
}

```

这个设计的实现原理是通过重写 `FilterInvocationSecurityMetadataSource` 的“`getAttributes`”方法，从用户访问的 URL 资源中检查系统的角色列表中是否存在互相匹配的权限设置，如果存在则将其存入一个安全元数据的角色列表之中，这个列表将为后面的权限检查提供依据。

从这里可以看出，如果一个资源我们不指定任何一个角色可以访问，那么这个资源对于所有用户来说都有访问权限。因为资源的元数据管理使用了动态加载的方法，所以对用户的权限管理也能实现在线更新，同时，这里也借助缓存技术来提高元数据的访问性能。安全资源的元数据管理是为用户权限检查做准备的。

一个用户对所访问的资源是否具有权限，使用了如下所示的权限检查设计：

```

@Override
public void decide(Authentication authentication, Object object,

```

```

        Collection<ConfigAttribute> configAttributes)
        throws AccessDeniedException, InsufficientAuthenticationException {
    if (configAttributes == null) {
        return;
    }

    //从 CustomSecurityMetadataSource (getAttributes) 中获取请求资源所需的角色集合
    Iterator<ConfigAttribute> iterator = configAttributes.iterator();

    while (iterator.hasNext()) {
        ConfigAttribute configAttribute = iterator.next();
        //对资源访问具有权限的角色
        String needRole = configAttribute.getAttribute();
        logger.info("具有权限的角色: " + needRole);
        //在用户拥有的权限中检查是否具有匹配的角色
        for (GrantedAuthority ga : authentication.getAuthorities()) {
            if (needRole.equals(ga.getAuthority())) {
                return;
            }
        }
    }
    //如果所有用户角色都不匹配, 则用户没有权限
    throw new AccessDeniedException("Cannot Access!");
}

```

这个设计的实现原理是通过重写 `AccessDecisionManager` 的权限决断方法“`decide`”，将安全管理元数据中的角色与用户的角色进行比较，如果用户的角色与元数据的角色匹配，就说明用户具有访问权限，否则就没有访问权限。

10.4.4 如何在应用中接入 SSO

完成 SSO 的客户端封装设计之后，在一个 Web 应用中使用 SSO 的功能就变得很简单了，首先只要在应用的项目管理配置中增加对 SSO 客户端的依赖引用，然后在应用配置中增加一些安全管理的配置，就可以实现使用 SSO 的功能。

下面以“`merchant-web`”模块为例进行说明，其他 Web UI 应用都可以参照这种方法接入 SSO。在本书提供的几个实例工程中，其他应用接入 SSO 的实例可以参考“`catalog-microservice`”工程的“`V1.1`”分支中模块“`catalog-web`”的实现方法。

首先，在项目配置管理中引用 SSO 客户端设计的封装包，如下所示：

```
<dependency>
```



```

<groupId>com.demo</groupId>
<artifactId>merchant-security</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>

```

然后，在应用的配置文件“application.yml”中使用类似于如下所示的设置：

```

security:
  ignored: /favicon.ico,/scripts/**,/styles/**,/images/**,/webjars/**
  sessions: ALWAYS
  OAuth2:
    sso:
      loginPath: /login
    client:
      clientId: ssoclient
      clientSecret: ssossecret
      accessTokenUri: http://localhost:8000/oauth/token
      userAuthorizationUri: http://localhost:8000/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      jwt:
        keyUri: http://localhost:8000/oauth/token_key

securityconfig:
  logoutsuccssurl: /tosignout
  permitall: /test/**,/hystrix**
  deniedpage: /deny
  ssoshome: http://localhost:8000/

```

其中，“security”及其“OAuth2”配置项是 Spring Security 和 OAuth2 组件所使用的一些配置参数，我们使用这些参数设置了“clientId”和“clientSecret”，即 SSO 服务端设计的配置类中设定的客户端 ID 和密钥。而“accessTokenUri”和“userAuthorizationUri”分别用来指定获取令牌和进行认证的端点设置。“jwt”的“keyUri”用来指定用于资源访问的公有密钥的端点设置。

“securityconfig”配置项下面的几个设置是由配置类 SecuritySettings 提供的几个自定义配置参数设定的。其中，“ssoshome”将为各个应用提供一个访问 SSO 首页的链接。

除了上面这些配置之外，对于接入了 SSO 的 Web 应用，在数据编辑和管理方面还需要做一些调整，以保证数据的创建和编辑能够正常提交。另外，接入了 SSO 的应用还可以实现根据用户权限自动分配菜单的功能。这两部分功能的实现请参考以下两个小节的说明。

10.4.5 有关跨站请求伪造防御的相关设置

使用了 Spring Security 之后，必须在页面中增加跨站请求伪造防御的相关设置，才能在创建或编辑数据时正常提交表单，否则有关表单提交的请求将会被拒绝访问。

首先，在页面中，我们统一在页面模板“layout.html”的头部增加了如下所示的设置：

```
<meta name="_csrf" th:content="${_csrf.token}"/>
<meta name="_csrf_header" th:content="${_csrf.headerName}"/>
```

然后，在一个公共调用的“public.js”中增加如下所示的设计以接收来自页面的传递参数：

```
$(function () {
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(header, token);
    });
});
```

这样做的目的可以让后台能够验证页面表单提交的合法性，从而对于数据提交起到了一定的保护作用。

10.4.6 根据用户权限自动分配菜单

通过登录用户的关联对象可以取得一个用户的菜单列表，这样就可以根据用户权限来自动分配用户的系统菜单。

首先，在应用的主页控制器设计中使用如下所示的实现方法：

```
@RequestMapping("/index")
public CompletableFuture<String> index(ModelMap model, Principal user,
HttpServletRequest request) throws Exception{
    return CompletableFuture.supplyAsync(() -> {
        List<ModelQo> menus = super.getModels(user.getName(), request);
        model.addAttribute("menus", menus);
        model.addAttribute("user", user);
        model.addAttribute("ssohome", ssohome);
        return "user/index";
    });
}
```

其中，“getModels”就是获取用户菜单的具体实现，其实现的代码如下所示：


```

@Autowired
private UserRestService userService;

@Value("${spring.application.name}")
private String serviceName;

public List<ModelQo> getModels(String userName, HttpServletRequest
request){
    //根据登录用户获取用户对象
    String json = userService.findByName(userName);
    UserQo user = new Gson().fromJson(json, UserQo.class);

    //根据匹配分类获取模块（二级菜单）列表
    List<ModelQo> modelList = new ArrayList<>();
    List<Long> modelIds = new ArrayList<>();
    for(RoleQo role : user.getRoles()){
        for(ResourceQo resource : role.getResources()){
            //分类顶级菜单链接
            String link = resource.getModel().getKind().getLink();
            //获取模块列表，去重
            if(! modelIds.contains(resource.getModel().getId())
                && pathMatcher.match(serviceName, link)){
                modelList.add(resource.getModel());
                modelIds.add(resource.getModel().getId());
            }
        }
    }

    return modelList;
}

```

即从用户的角色中找出其关联的资源列表，再从每一个资源中找出与当前应用的名称互相匹配的模块对象，然后经过去重整理之后得到的模块列表就是一个应用的菜单体系。

使用这个模块列表，在导航页面上使用如下所示的设计循环输出菜单：

```

<ul >
    <li th:each="model:${menus}"><a th:classappend="${page == '${model.
host}}' ? 'currentPageNav' : ''" th:href="@{${model.host}}" th:text="${model.
name}"></a></li>
</ul>

```

完成上面的开发之后，可以开始进行测试。确认注册中心已经启动，然后分别启动“merchant-restapi”应用、“merchant-ssso”应用和“merchant-web”应用。

所有应用启动成功之后，通过浏览器打开如下商家系统 Web 应用的链接：

`http://localhost:8081`

打开链接后，在出现的登录界面中输入上面单元测试中生成的用户名和密码进行登录。登录成功后就可以打开商家系统“merchant-web”应用的主页。

商家系统只有一个用户管理的功能，所以它的主页显示的效果如图 10-6 所示。在这里，商家管理员可以进行用户管理的操作。

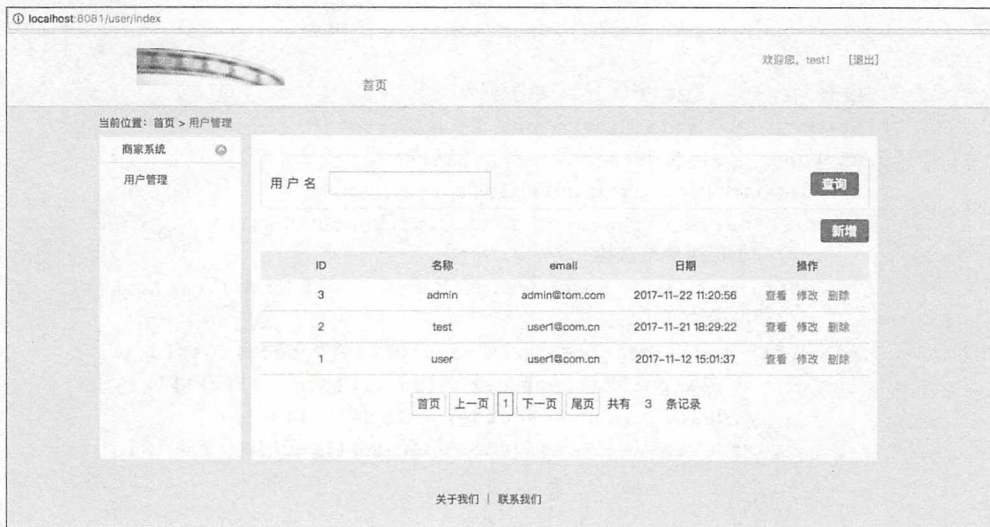


图 10-6 商家系统主页

10.5 小结

本章通过商家权限体系和 SSO 的设计，构造了一个安全可靠的商家管理后台。在商家管理后台中，商家用户通过统一的权限管理，可以使用在分布式环境之中任何其他已经接入 SSO 的微服务应用。商家管理后台的设计以一种更加完善的方式，将各个分散开发的微服务组成了一个功能丰富的整体，充分体现了微服务架构设计中的强大优势。

其中，商家权限体系的设计以访问资源为基础建立了三级菜单体系，并通过角色与资源的关系，将用户权限与菜单组成一个有机的整体。但对于商家的角色及其菜单的管理配置，必须由平台运营方进行操作，在下一章的平台管理后台的开发中将实现对商家的权限配置进行管理的功能。

11

平台管理后台开发

平台管理后台是为电商平台的运营方提供服务的,它主要包含商家管理和一些公共配置管理的功能。在商家管理的设计中,包括商家的注册、审核和商家用户的权限管理,以及菜单配置管理等功能。除了一些公共管理功能的设计之外,平台本身的安全管理设计也是一项开发的内容。另外,有关商品类目的管理也应该纳入平台的管理范畴之中,这样对整个平台类目设置才能有一个统一的规范。

平台管理后台的项目工程为“**manage-microservice**”,完整的源代码可以从如下链接取得:

<https://gitee.com/chenshaojian/manage-microservice.git>

平台管理后台的开发主要包含两大部分的内容:一部分是管理后台本身的权限管理设计;另一部分是商家及其权限的管理。

11.1 平台管理后台领域设计

平台管理后台是一个独立的应用系统,它有自身的用户体系和独立的权限管理设计。平台管理后台的业务领域设计在模块“**manage-domain**”中进行,管理后台的领域模型主要由操作员、角色和部门等实体所组成,这几个实体的关系是,一个操作员只能从属于一个部门,同时一个操作员可以拥有多个角色。

11.1.1 领域实体建模

操作员是平台管理的用户,操作员的实体设计如下所示:

```

@Entity
@Table(name = "t_operator")
public class Operators extends IdEntity implements java.io.Serializable{
    private String name;
    private String email;
    private Integer sex;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;
    private String password;

    @ManyToOne
    @JoinColumn(name = "did")
    @JsonBackReference
    private Department department;

    @ManyToMany(cascade = {}, fetch = FetchType.EAGER)
    @JoinTable(name = "operator_part",
        joinColumns = {@JoinColumn(name = "operator_id")},
        inverseJoinColumns = {@JoinColumn(name = "part_id")})
    private List<Part> parts = new ArrayList<>();

    public Operators() {
    }
    .....
}

```

操作员一方面以多对一的关系关联了部门实体，即一个操作员只能属于一个部门；另一方面以多对多的关系关联了角色实体，即一个操作员可以拥有多个角色。

其中，通过继承 `IdEntity` 取得了实体的 ID 属性，`IdEntity` 的设计如下所示：

```

@MappedSuperclass
public abstract class IdEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    public Long getId() {
        return id;
    }
}

```




```

    }

    public void setId(Long id) {
        this.id = id;
    }
}

```

操作员的权限将由所拥有的角色来决定，角色实体的设计如下所示：

```

@Entity
@Table(name = "t_part")
public class Part extends IdEntity implements java.io.Serializable{
    private String name;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default
current_timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    public Part() {
    }
    .....
}

```

这个角色设计并没有关联资源，有关它的访问权限设计将在后面使用一种更加简单的方式来实现。

部门实体的设计如下所示：

```

@Entity
@Table(name = "t_department")
public class Department extends IdEntity implements java.io.Serializable{
    private String name;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    @Column(name = "created", columnDefinition = "timestamp default current_
timestamp")
    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    public Department() {
    }
    .....
}

```

在实体之间的关联关系设计中，使用了单向关联的设计原则，所以在部门实体设计中



并不再与操作员建立关联关系，如果要从部门中查询操作员可以通过使用 SQL 查询来实现。

11.1.2 实体的行为设计

实体通过绑定 JPA 的存储库就可以为之赋予一些基本行为，而在存储库接口的定义之中，我们还可以通过声明方法来增加一个实体的操作行为。其中，有关操作员实体的行为设计如下所示：

```
@Repository
public interface OperatorRepository extends JpaRepository<Operators, Long>,
JpaSpecificationExecutor<Operators> {
    @Query("select t from Operators t where t.name =?1 and t.email =?2")
    Operators findByNameAndEmail(String name, String email);

    @Query("select distinct u from Operators u where u.name= :name")
    Operators findByName(@Param("name") String name);

    @Query("select distinct u from Operators u where u.id= :id")
    Operators findById(@Param("id") Long id);

    @Query("select o from Operators o " +
        "left join o.parts p " +
        "where p.id= :id")
    List<Operators> findByPartId(@Param("id") Long id);
}
```

这里声明了几个方法，都是通过 SQL 查询语句的使用来扩展操作员实体的操作行为的，其中的“findByPartId”实现了通过部门 ID 来查询操作员列表的功能。

另外，角色实体和部门实体的行为设计只要通过简单的接口绑定就可以实现，不再赘述。

11.1.3 领域服务开发

领域服务是对存储库接口调用的一个服务层封装设计，通过服务层的开发可以为存储库接口的调用提供统一的事务管理，以及实现其他扩展设计。下面只以操作员实体的领域服务开发为例进行说明。

如下所示是一些增、删、改、查的实现方法，在这些方法的调用中都增加了使用缓存的功能：




```
@Service
@Transactional
public class OperatorService {
    @Autowired
    private OperatorRepository operatorRepository;
    @Autowired
    private CacheComponent cacheComponent;

    public void save(Operators operators){
        //删除缓存
        if(!StringUtils.isEmpty(operators.getId())){
            String key = operators.getId().toString();
            //删除原有缓存
            cacheComponent.remove(Constant.BOSS_BACKEND_OPERATOR_ID, key);
        }
        operatorRepository.save(operators);
        //保存缓存
        if(!StringUtils.isEmpty(operators.getId())){
            String key = operators.getId().toString();
            cacheComponent.put(Constant.BOSS_BACKEND_OPERATOR_ID, key,
operators, 12); //增加缓存, 保存 12 秒
        }
    }

    public void delete(Long id){
        //删除缓存
        cacheComponent.remove(Constant.BOSS_BACKEND_OPERATOR_ID, id.
toString());
        operatorRepository.delete(id);
    }

    public List<Operators> findAll(){
        return operatorRepository.findAll();
    }

    public Operators findOne(Long id){
        Operators operators = null;
        //使用缓存
        Object object = cacheComponent.get(Constant.BOSS_BACKEND_OPERATOR_ID,
id.toString());
        if (CommonUtils.isNull(object)) {
            operators = operatorRepository.findById(id);
            if (operators != null)
                cacheComponent.put(Constant.BOSS_BACKEND_OPERATOR_ID, id.
```



```

toString(), operators, 12);
    } else {
        operators = (Operators) object;
    }
    return operators;
}
}

```

这些方法都是通过调用操作员实体的存储库接口实现了数据存取的操作。其中，缓存的使用要注意读写顺序的安排及其有效期的设定。

如下所示是一个分页查询的设计：

```

public Page<Operators> findAll(OperatorsVo operatorsVo) {
    Sort sort = new Sort(Sort.Direction.DESC, "created");
    Pageable pageable = new PageRequest(operatorsVo.getPage(),
operatorsVo.getSize(), sort);

    return operatorRepository.findAll(new Specification<Operators>() {
        @Override
        public Predicate toPredicate(Root<Operators> root, CriteriaQuery
<?> query, CriteriaBuilder criteriaBuilder) {
            List<Predicate> predicatesList = new ArrayList<Predicate>();

            if (CommonUtils.isNotNull(operatorsVo.getName())) {
                predicatesList.add(criteriaBuilder.like(root.get("name"),
%" + operatorsVo.getName() + "%"));
            }
            if (CommonUtils.isNotNull(operatorsVo.getCreated())) {
                predicatesList.add(criteriaBuilder.greaterThan(root.get
("created"), operatorsVo.getCreated()));
            }

            query.where(predicatesList.toArray(new Predicate[predicates
List.size()]));

            return query.getRestriction();
        }
    }, pageable);
}

```

其中，分页查询的参数可以根据需要进行设计，这里只提供了操作员名称和创建日期两个参数进行查询。



11.1.4 领域服务单元测试

对领域业务模型所提供的服务进行单元测试，可以在模块“manage-restapi”中进行，在这里所进行的测试与实际中进行调用的方式是完全一样的。

在执行测试之前，必须在工程的配置文件“application.yml”中配置好数据库、JPA 和缓存的各项参数。有关这些配置在前面的章节中已经有过相关的描述和介绍，这里不一样的地方就是数据库的名字。我们把平台管理的数据库名字设为“managedb”，在测试之前请确定已经创建了这个数据库，并且设置相关的用户具有访问权限。配置完成之后，我们可以使用下面的测试用例进行测试。

首先，可以执行创建实体对象的测试，如下所示：

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {JpaConfiguration.class, ManageRestApi
Application.class})
@SpringBootTest
public class BbServiceTest {
    private static Logger logger = LoggerFactory.getLogger(BbServiceTest.
class);

    @Autowired
    private OperatorService operatorService;
    @Autowired
    private PartService partService;
    @Autowired
    private DepartmentService departmentService;

    @Test
    public void insertData(){
        Part part = new Part();
        part.setName("admins");

        partService.save(part);

        Department department = new Department();
        department.setName("技术部");
        departmentService.save(department);

        Operators operators = new Operators();
        operators.setName("admin");
        operators.setSex(1);
```



```

        operators.setDepartment(department);

        List<Part> partList = operators.getParts();
        partList.add(part);
        operators.setParts(partList);

        BCryptPasswordEncoder bc = new BCryptPasswordEncoder();
        operators.setPassword(bc.encode("123456"));

        operatorService.save(operators);
        assert operators.getId() > 0 : "create error";
    }
}

```

这个测试用例执行了如下一些操作：

- 创建了一个角色，名称为“admins”（也可以把它看成是一个管理员组）。
- 创建了一个部门，名称为“技术部”。
- 创建了一个用户，名称为“admin”，并为用户设定了密码“123456”。

如果测试成功通过，这些生成的数据将可以为后面的开发所使用，我们将使用“admin”这个用户作为系统的管理员来登录系统。其他测试用例，可以参照这个方法来进行设计。

11.2 平台管理后台访问控制设计

这里的访问控制设计还是使用 Spring Security 来实现，所以其中的大部分设计与上章的 SSO 设计中的实现方法很类似，不同的地方是这里并不需要 OAuth2，而且对权限管理的设计也使用了一种更为简单的方法来实现。下面我们略过一些相同的地方，只针对不同点进行说明。这些设计主要在模块“manage-web”中实现。

11.2.1 使用平台管理的用户体系

首先，为了在 Spring Security 的安全管理中采用平台的用户体系，通过继承 Operators 创建一个安全用户 SecurityUser，实现方法如下所示：

```

public class SecurityUser extends Operators implements UserDetails
{

    private static final long serialVersionUID = 1L;

    public SecurityUser(Operators user) {

```




```

        if (user != null)
        {
            this.setId(user.getId());
            this.setName(user.getName());
            this.setEmail(user.getEmail());
            this.setPassword(user.getPassword());
            this.setSex(user.getSex());
            this.setCreated(user.getCreated());
            this.setParts(user.getParts());
        }
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Collection<GrantedAuthority> authorities = new ArrayList<Granted
Authority>();
        for (Part role : this.getParts()) {
            SimpleGrantedAuthority authority = new SimpleGrantedAuthority
(role.getName());
            authorities.add(authority);
        }
        return authorities;
    }
    ...
}

```

即通过实现 Spring Security 的 UserDetails 来加载我们自定义的操作员实体对象 Operators，并使用操作员的角色进行权限设置。

然后，通过自定义一个 CustomUserDetailsService 实现 Spring Security 的 UserDetailsService，如下所示：

```

@Component
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired
    private OperatorService operatorService;

    @Override
    public UserDetails loadUserByUsername(String userName) throws UsernameNot
FoundException {
        Operators user = operatorService.findByName(userName);
        if (user == null) {
            throw new UsernameNotFoundException("UserName " + userName + " not
found");
        }
    }
}

```

```

    }
    return new SecurityUser(user);
  }
}

```

这样，当导入用户时，即可通过操作员的服务组件 `OperatorService` 导入操作员对象的用户体系信息，返回给上面定义的 `SecurityUser` 之中。

11.2.2 权限管理设计

这里的权限管理使用一种比较简单的方法来实现，即通过使用配置来实现权限管理，其实现方法说明如下。

首先，在模块的应用配置中增加如下所示的配置项：

```

securityconfig:
  logoutsuccssurl: /
  permitall: /druid/**,/bbs**
  deniedpage: /deny
  urlroles: /**/new/** = admins;
            /**/edit/** = admins,editors;
            /**/delete/** = admins

```

这些配置参数由我们自定义的一个配置类 `SecuritySettings` 实现。

其中的“urlroles”即为权限管理的配置参数，这个配置参数通过请求的 URL 来设定用户的访问权限。这里只设置了两个角色（或者说用户组）的权限，它们分别是“admins”和“editors”。其中 URL 资源配置中结合通配符“*”使用了几个关键字来指定，分别由“new”、“edit”和“delete”来表示新建、编辑和删除等操作。

这样，在控制器的设计中同样也要使用这几个关键字来设置 URL，例如，类似于如下所示的一些“`@RequestMapping`”设计：

```

@RequestMapping("/new")
@RequestMapping(value="/edit/{id}")
@RequestMapping(value="/update", method = RequestMethod.POST)
@RequestMapping(value="/delete/{id}")

```

然后，在安全资源管理的元数据管理中使用如下所示的设计：

```

public CustomSecurityMetadataSource (String urlroles) {
    super();
    this.urlroles = urlroles;
    resourceMap = loadResourceMatchAuthority();
}

```



```

    }

    private Map<String, Collection<ConfigAttribute>> loadResourceMatch
    Authority() {

        Map<String, Collection<ConfigAttribute>> map = new HashMap<String,
        Collection<ConfigAttribute>>();

        if(urlroles != null && !urlroles.isEmpty()){
            String[] resouces = urlroles.split(";");
            for(String resource : resouces){
                String[] urls = resource.split("=");
                String[] roles = urls[1].split(",");
                Collection<ConfigAttribute> list = new ArrayList<Config
Attribute>();
                for(String role : roles){
                    ConfigAttribute config = new SecurityConfig(role.trim());
                    list.add(config);
                }
                //key: url, value: roles
                map.put(urls[0].trim(), list);
            }
        }else{
            logger.error("'securityconfig.urlroles' must be set");
        }

        logger.info("Loaded UrlRoles Resources.");
        return map;
    }

```

这个设计表示当系统启动时即导入上面权限配置的数据,作为安全管理的元数据给后面的权限检查提供依据。

最后,在权限检查的设计中,使用如下所示的设计:

```

@Override
public void decide(Authentication authentication, Object object,
    Collection<ConfigAttribute> configAttributes)
    throws AccessDeniedException, InsufficientAuthenticationException {
    if (configAttributes == null) {
        return;
    }

    //config urlroles

```

```

        Iterator<ConfigAttribute> iterator = configAttributes.iterator();

        while (iterator.hasNext()) {
            ConfigAttribute configAttribute = iterator.next();
            //need role
            String needRole = configAttribute.getAttribute();
            //user roles
            for (GrantedAuthority ga : authentication.getAuthorities()) {
                if (needRole.equals(ga.getAuthority())) {
                    return;
                }
            }
            logger.info("need role is " + needRole);
        }
        throw new AccessDeniedException("Cannot Access!");
    }
}

```

当用户访问的资源中包含安全管理的元数据时,即检查用户的角色列表中是否有与之匹配的角色,以此达到权限验证的目的。这种简化的设计同时也要求我们在创建角色时,其名字必须与配置中的名字相匹配。完成上面所有设计后,就可以开始进行测试。

直接启动“manage-web”应用,启动成功之后,输入如下所示的链接登录系统:

<http://localhost:8099>

使用上面单元测试时生成的用户“admin”即可登录系统。登录系统后可以对操作员及其角色等数据进行管理,如图 11-1 所示。

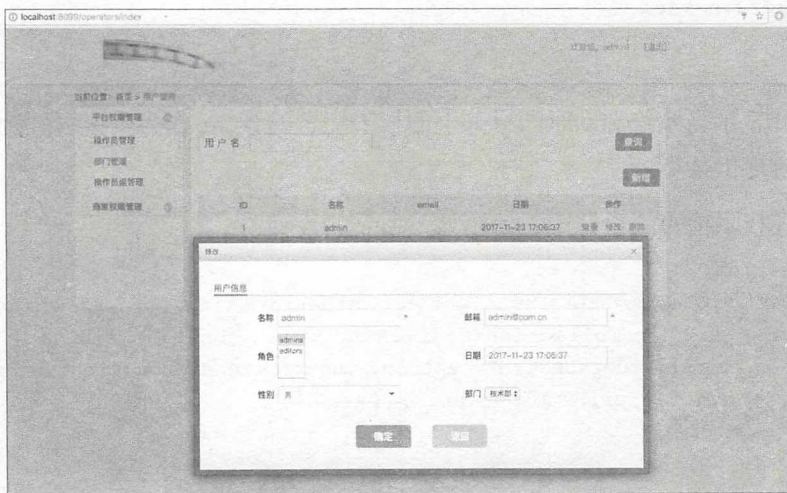


图 11-1 平台管理后台操作员管理

完成了管理后台本身的安全设计之后,下面介绍一下怎么在管理后台中实现对商家及其权限管理的功能。

11.3 商家的注册设计

对于商家管理的设计主要是通过调用商家服务的 Rest API 来实现的。注册一个商家时,除了创建一个商家对象之外,还必须为商家创建一个用户,这样商家才能使用这个用户来登录商家管理后台。

我们在商家的查询对象设计中增加两个字段用来增加商家所属用户的简要信息,如下所示:

```
public class MerchantQo extends PageQo implements Serializable {
    private Long id;
    private String name;
    private String email;
    private String phone;
    private String address;
    private String linkman;
    @DateTimeFormat(
        pattern = "yyyy-MM-dd HH:mm:ss"
    )
    private Date created;
    private String userName;
    private String passWord;

    public MerchantQo() {
    }
    .....
}
```

其中的“userName”和“passWord”分别用来表示商家用户的用户名和密码。这样在新建商家的页面设计中,就可以使用如下所示的表单设计:

```
<form id="saveForm" action="./save" method="post">
    <table class="addNewInfList">
        <tr>
            <th>名称</th>
            <td width="240">
                <input class="inp-list w-200 clear-mr f-left" type="text"
id="name" name="name" maxlength="32" />
```

```

        <span class="tipStar f-left">*</span>
    </td>
    <th>邮箱</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="email" name="email" maxlength="128" />
    </td>
</tr>
<tr>
    <th>电话</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="phone" name="phone" maxlength="32" />
        <span class="tipStar f-left">*</span>
    </td>
    <th>地址</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="address" name="address" maxlength="255" />
    </td>
</tr>
<tr>
    <th>联系人</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="linkman" name="linkman" maxlength="128" />
    </td>
    <th>日期</th>
    <td>
        <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" id="created" name="created"/>
    </td>
</tr>
<tr>
    <th>用户名</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="userName" name="userName" maxlength="128" />
        <span class="tipStar f-left">*</span>
    </td>
    <th>密码</th>
    <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="password"
id="passWord" name="passWord" maxlength="32" />

```



```

        <span class="tipStar f-left">*</span>
    </td>
</tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog()">返回</a>
</div>
</form>

```

完成设计后，新建商家的设计效果如图 11-2 所示。

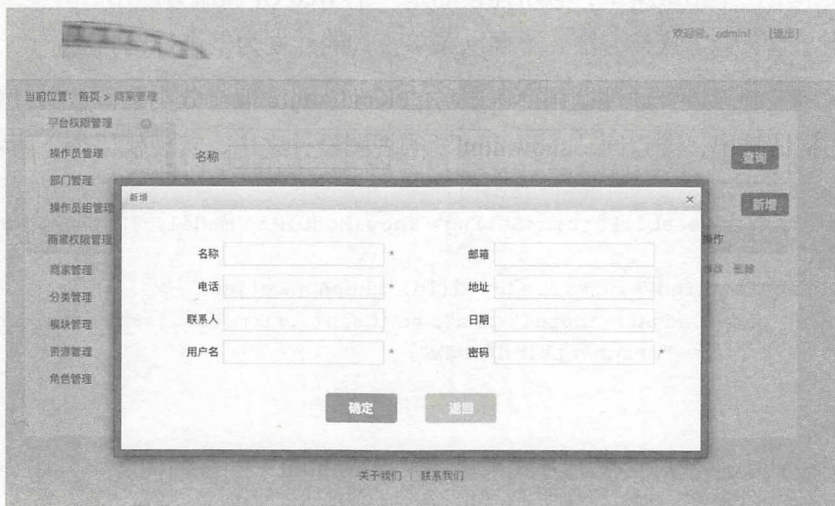


图 11-2 新建商家的操作界面

当新增商家提交表单时，将通过控制器实现对商家服务的 `MerchantFuture` 进行远程调用，实现的代码如下所示：

```

@RequestMapping(value="/save", method = RequestMethod.POST)
@ResponseBody
public CompletableFuture<String> save(MerchantQo merchantQo,
    HttpServletRequest request) throws Exception{
    return merchantFuture.create(merchantQo).thenApply(sid -> {
        logger.info("新增->ID=" + sid);
        return sid;
    });
}

```

这里直接使用查询对象 `MerchantQo` 来传递参数，商家服务在接收到请求后将会创建一个商家和一个商家用户，并默认为其分配管理员角色。

11.4 商家菜单体系管理开发

在商家的菜单体系中我们设计了三级菜单，分别为分类、模块和资源。在平台管理后台中，必须实现对这些菜单进行统一管理。下面对各个菜单的管理如何进行开发分别加以说明。

11.4.1 分类菜单管理开发

分类菜单是一个顶级菜单，它所连接的是一个 Web UI 微服务应用。分类菜单的管理包括增、删、改、查等操作方法，下面以菜单查询的开发为例进行说明。

首先使用控制器设计通过调用商家服务的 `KindFuture` 取得分类菜单数据，然后将查询结果转换为视图输出，即返回“`show.html`”的视图设计之中，实现代码如下所示：

```
@RequestMapping(value="/{id}")
public CompletableFuture<String> show(ModelMap model, @PathVariable Long
id) {
    return kindFuture.findById(id).thenApply(json -> {
        model.addAttribute("kind", new Gson().fromJson(json, KindQo.class));
        return "merchantkind/show";
    });
}
```

在视图设计中通过对话框的方式来显示“`show.html`”的页面内容，其中，页面设计部分的实现代码如下所示：

```
<div class="addInfBtn">
    <h3 class="itemTit"><span>分类信息</span></h3>
    <table class="addNewInfList">
        <tr>
            <th>名称</th>
            <td width="240"><input class="inp-list w-200 clear-mr f-left"
type="text" th:value="{kind.name}" readonly="true"/></td>
            <th>链接</th>
            <td width="240">
                <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="{kind.link}" readonly="true" />
            </td>
        </tr>
        <tr>
            <th>日期</th>
            <td>
```



```

        <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="{kind.created} ?
${#dates.format(kind.created,'yyyy-MM-dd HH:mm:ss')}" : ''" />
    </td>
</tr>
</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回
</a>
</div>
</div>

```

完成设计后的显示效果如图 11-3 所示, 所显示的内容是一个“订单系统”的分类菜单的查询信息, 其中“链接”使用的是订单微服务应用的实例名称。

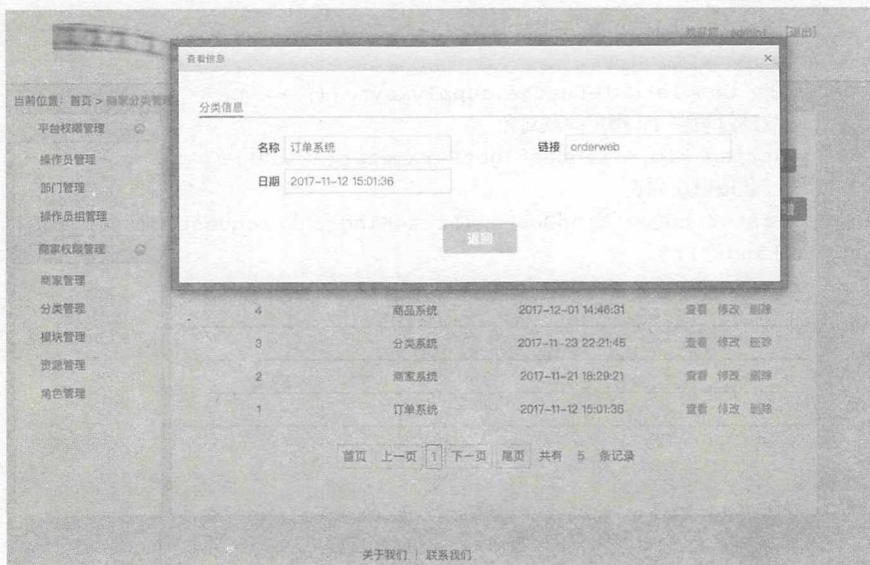


图 11-3 分类菜单管理

11.4.2 模块菜单管理开发

模块菜单是商家管理后台的一个二级菜单, 它表示一个应用中的一个业务类型, 例如, 在订单系统中可以具有订单管理和订单报表等模块菜单。

模块菜单管理包括菜单的增、删、改、查等方面的操作内容, 下面以新建模块菜单的开发为例进行说明。如下所示是一个新建菜单的控制器设计:

```

    @RequestMapping("/new")
    public CompletableFuture<String> create(ModelMap model, HttpServletRequest
Request request){
        return kindFuture.findList().thenApply(json -> {
            List<KindQo> kindQos = new Gson().fromJson(json, new TypeToken<List
<KindQo>>() {}.getType());
            //缓存模块列表为 save 方法使用
            request.getSession().setAttribute("kinds", kindQos);
            model.addAttribute("kinds", kindQos);
            return "merchantmodel/new";
        });
    }

    @RequestMapping(value="/save", method = RequestMethod.POST)
    @ResponseBody
    public CompletableFuture<String> save(ModelQo modelQo, HttpServletRequest
request) throws Exception{
        return CompletableFuture.supplyAsync(() -> {
            //通过模块 ID 指定关联对象
            String kid = request.getParameter("kid");
            //获取模块列表
            List<KindQo> kindQos = (List<KindQo>) request.getSession().
getAttribute("kinds");
            for (KindQo kindQo : kindQos) {
                if (kindQo.getId().compareTo(new Long(kid)) == 0) {
                    modelQo.setKind(kindQo);
                    break;
                }
            }

            String sid = modelRestService.create(modelQo);
            logger.info("新增->ID=" + sid);
            return sid;
        });
    }
}

```

需要注意的是，这里使用了查询对象“ModelQo”来获取表单的参数，这与使用实体对象来获取参数时是有些不一样的，即使用查询对象时将不能得到所关联的对象，所以这里使用了“kid”这个字符串参数来表示模块所关联的分类对象的 ID，然后从我们在会话中保存的对象列表中取得相关对象，而不是使用“kind”这样的参数直接取得所关联的分类对象。

这样，在相关页面视图设计上，也必须要有与之对应的设计。如下所示是一个新建模

块菜单的表单设计:

```
<form id="saveForm" action="./save" method="post">
  <table class="addNewInfList">
    <tr>
      <th>名称</th>
      <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
name="name" id="name" maxlength="32" />
        <span class="tipStar f-left">*</span>
      </td>
      <th>URL</th>
      <td width="240">
        <input class="inp-list w-200 clear-mr f-left" type="text"
id="host" name="host" maxlength="64" />
      </td>
    </tr>
    <tr>
      <th>日期</th>
      <td>
        <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" id="created" name="created"/>
      </td>
      <th>分类</th>
      <td width="240">
        <div >
          <select name="kid" id="kid">
            <option th:each="kind:${kinds}" th:value="${kind.id}"
              th:text="${#strings.length(kind.name)>20?
#strings.substring(kind.name,0,20)+'...':kind.name}"
              ></option>
          </select>
          <span class="tipStar f-right">*</span>
        </div>
      </td>
    </tr>
  </table>
  <div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog()">返回</a>
  </div>
</form>
```

其中, 模块菜单关联的分类所使用的“select”控件中, 使用了“kid”来取得分类对象的 ID。

完成设计后, 模块菜单管理的显示效果如图 11-4 所示, 这是一个“订单管理”的模块菜单的编辑操作界面, 其中的 URL 是进入订单管理主页的一个链接地址, 菜单所关联的上级菜单为“订单系统”。

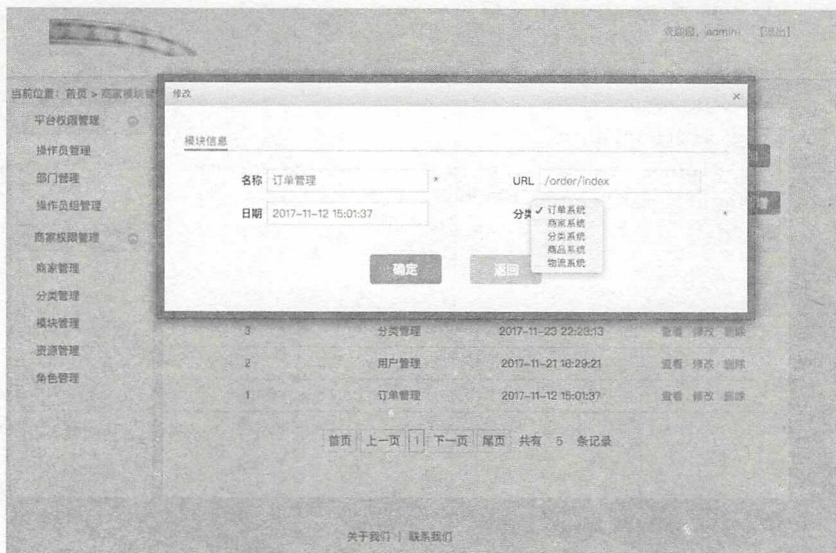


图 11-4 模块菜单管理

11.4.3 访问资源管理开发

访问资源是商家管理后台的一个三级菜单, 例如, 对于模块菜单“订单管理”来说, 它可以具有“订单修改”和“订单删除”等子菜单。访问资源是最小的一个权限管理单元, 在权限管理设计中它是角色所关联的访问对象。访问资源的管理包括增、删、改、查等方面的操作内容, 下面以资源的编辑开发为例进行说明。

如下所示是一个资源编辑的控制器设计:

```
@RequestMapping("/edit/{id}")
public CompletableFuture<String> edit(@PathVariable Long id, ModelMap
model, HttpServletRequest request) {
    return CompletableFuture.supplyAsync(() -> resourceRestService.
findById(id))
        .thenCompose(json -> CompletableFuture.supplyAsync(() -> {
```



```

        ResourceQo resourceQo = new Gson().fromJson(json, ResourceQo.class);

        String models = modelRestService.findList();
        List<ModelQo> modelQoList = new Gson().fromJson(models, new
TypeToken<List<ModelQo>>() {}.getType());

        //缓存模块列表为 update 方法使用
        request.getSession().setAttribute("models", modelQoList);

        model.addAttribute("models", modelQoList);
        model.addAttribute("resource", resourceQo);

        return "merchantresource/edit";
    });
}

@RequestMapping(method = RequestMethod.POST, value="/update")
@ResponseBody
public CompletableFuture<String> update(ResourceQo resourceQo,
HttpServletRequest request) throws Exception{
    return CompletableFuture.supplyAsync(() -> {
        //通过模块 ID 指定关联对象
        String mid = request.getParameter("mid");
        //获取模块列表
        List<ModelQo> modelQos = (List<ModelQo>) request.getSession().
getAttribute("models");
        for (ModelQo modelQo : modelQos) {
            if (modelQo.getId().compareTo(new Long(mid)) == 0) {
                resourceQo.setModel(modelQo);
                break;
            }
        }

        String sid = resourceRestService.update(resourceQo);
        logger.info("修改->ID=" + sid);
        return sid;
    });
}

```

在进行资源编辑之前，首先取出模块列表，使用这个模块列表在页面中设计一个下拉框。然后在资源编辑时从这个下拉框中选择一个模块以设置它的关联关系。最后，在数据

保存时，通过模块 ID 取出相应的对象进行保存。其对应的页面设计如下所示：

```
<script th:src="@{/scripts/merchantresource/edit.js}"></script>
<form id="saveForm" method="post">
  <input type="hidden" name="id" id="id" th:value="${resource.id}"/>
  <div class="addInfBtn" >
    <h3 class="itemTit"><span>资源信息</span></h3>
    <table class="addNewInfList">
      <tr>
        <th>名称</th>
        <td width="240">
          <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="${resource.name}" id="name" name="name" maxlength="32" />
          <span class="tipStar f-left">*</span>
        </td>
        <th>模块</th>
        <td width="240">
          <div >
            <select name="mid" id="mid">
              <option th:each="model:${models}" th:value="${model.id}"
th:text="${#strings.length(model.name)>20?
#strings.substring(model.name,0,20)+'...':model.name}"
th:selected="${resource.model !=null and
resource.model.id == model.id}"
></option>
            </select>
            <span class="tipStar f-right">*</span>
          </div>
        </td>
      </tr>
      <tr>
        <th>URL</th>
        <td width="240">
          <input class="inp-list w-200 clear-mr f-left" type="text"
th:value="${resource.url}" id="url" name="url" maxlength="64" />
        </td>
        <th>日期</th>
        <td>
          <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="${resource.
created} ? ${#dates.format(resource.created,'yyyy-MM-dd HH:mm:ss')}" : ""
id="created" name="created"/>
        </td>
      </tr>
    </table>
  </div>
</form>
```



```

</table>
<div class="bottomBtnBox">
    <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
    <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回
</a>
</div>
</div>
</div>

</form>

```

完成设计后资源管理的显示效果如图 11-5 所示。

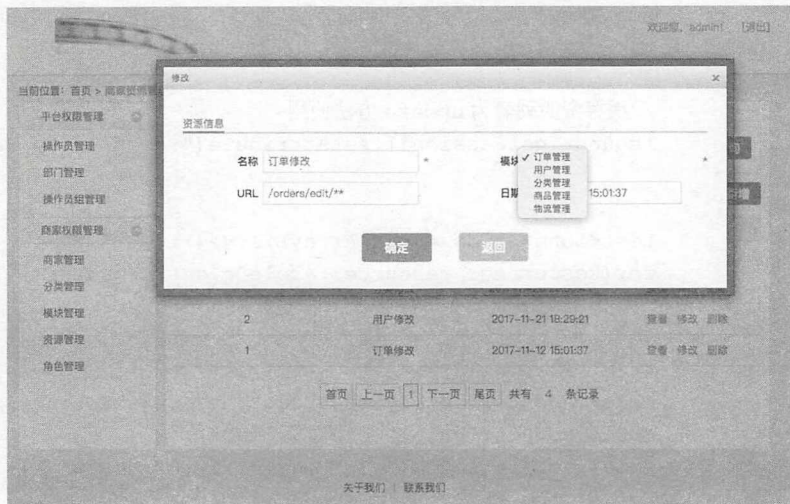


图 11-5 资源管理

如图 11-5 所示为一个“订单修改”的资源管理的操作界面，其中，URL 是一个执行订单修改的链接地址，链接地址后面增加几个符号“/**”是为了方便权限的检查，也可以省略不用，而所关联的模块中的订单管理就是订单修改的上级菜单。

11.5 商家角色管理开发

商家的权限管理通过角色管理来实现，角色与资源建立关联关系表示角色对该资源具有访问权限，而一个用户拥有哪些角色，就表示这个用户对这些角色所关联的资源具有访问权限。

角色管理主要通过 RoleFuture 等组件访问商家服务提供的接口，从而实现对角色的数

据配置进行管理。角色管理包括角色的增、删、改、查等方面的操作内容,下面以角色修改的开发为例进行说明。如下所示是进行角色修改时的控制器设计:

```
@RequestMapping("/edit/{id}")
public CompletableFuture<String> edit(@PathVariable Long id, ModelMap
model, HttpServletRequest request) {
    return CompletableFuture.supplyAsync(() -> roleRestService.findById(id))
        .thenCompose(json -> CompletableFuture.supplyAsync(() -> {
            RoleQo roleQo = new Gson().fromJson(json, RoleQo.class);

            String resources = resourceRestService.findList();
            List<ResourceQo> resourceVoList = new Gson().fromJson
(resources, new TypeToken<List<ResourceQo>>() {}).getType());

            //缓存资源列表为 update 方法使用
            request.getSession().setAttribute("resources", resource
VoList);

            List<Long> rids = new ArrayList<>();
            for(ResourceQo resource : roleQo.getResources()){
                rids.add(resource.getId());
            }

            model.addAttribute("resources", resourceVoList);
            model.addAttribute("role", roleQo);
            model.addAttribute("rids", rids);

            return "merchantrole/edit";
        }));
}
```

其中,对于角色已经关联的资源使用一个由资源 ID 构成的列表“rids”,可以在页面的多项选择下拉列表框中对已经选择的资源做出判断,如果是已经关联的角色就设定为选中的样式。对应的页面设计的实现代码如下所示:

```
<td width="240">
    <div >
        <select name="rids" id="rids" multiple="multiple">
            <option th:each="resource:${resources}" th:value=
"$ {resource.id}"
                                th:text="${#strings.length(resource.name)>20?
#strings.substring(resource.name,0,20)+'...':resource.name}"
                                th:selected="${#lists.contains(rids,
resource.id)}">
```



```

        </option>
    </select>
</div>
</td>

```

其中，使用“lists”函数来判断下拉列表框的资源是否已经被角色所关联，如果是则选中样式设置为“true”。完成设计后，显示的效果如图 11-6 所示。

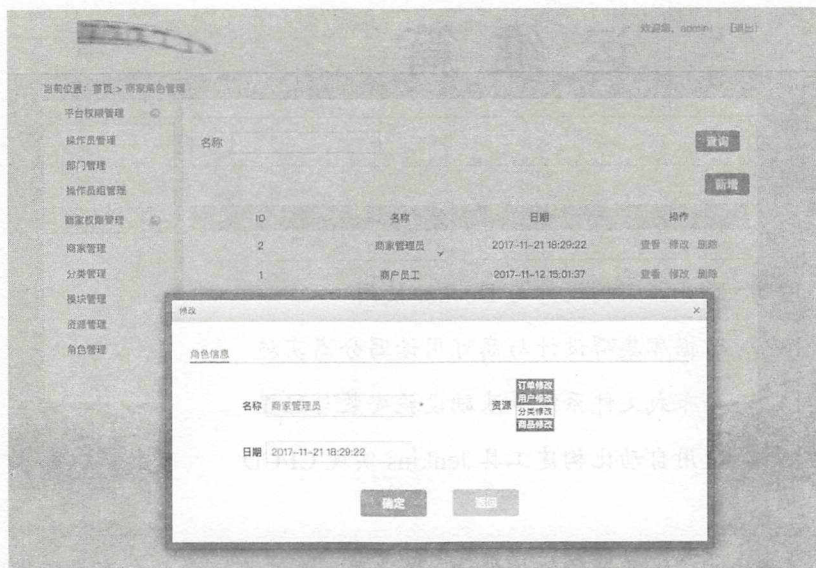


图 11-6 角色管理

11.6 小结

本章主要实现了平台管理后台的访问控制设计、商家注册及其权限和菜单管理等方面的功能。其中，商家注册及其权限、菜单的配置和管理都是通过调用商家服务的 Rest API 接口来实现的。实际上，在我们的微服务架构设计中，Web UI 微服务的开发都是通过调用 Rest API 来进行的，所以在平台管理后台中要实现对其他服务的管理是非常容易的。

有关微服务的开发至此告一段落，从下一章开始，我们将从运维的角度来探讨微服务的部署及其微服务运行环境的构建等方面的内容。

第三部分

运维篇

第 12 章 服务器架构设计与 Docker 使用

第 13 章 数据库集群设计与高可用读写分离实施

第 14 章 分布式文件系统等基础设施安装与配置

第 15 章 使用自动化构建工具 Jenkins 实现 CI/CD

这一部分通过服务器的架构设计，搭建了一个安全可靠的分布式环境，在此环境中，进行了数据库集群和分布式文件系统等基础服务的安装和设置，并演示了使用 Docker 进行微服务发布和使用 Jenkins 实现自动部署的方法。

服务器架构设计与 Docker 使用

完成微服务的开发，最终的部署和发布必须为其提供一个合适的分布式环境，才能充分发挥微服务架构的优势。这个环境首先应该是安全可靠的，并且可以进行任意扩展的分布式环境。然后，它的基础设施也应该是配备齐全，并且稳定可靠和可扩展的。这些基础设施包括数据库管理系统、文件管理系统、消息服务系统等服务，以及自动测试和持续交付等工具的支持。

我们开发的每一个微服务都可以进行任意多副本的发布，能够持续保持高性能的服务状态，所以微服务应用的基础服务设施和构建环境，都必须具有可持续扩展的特性。

本章先介绍服务器架构设计及其组建的方法，以及怎样使用 **Docker** 来构建一个高可用的微服务治理环境。其他各个方面的内容将在后续的章节中分别进行详细说明。

12.1 服务器组建

一个应用平台将会拥有很多微服务应用，它必须有一定规模的服务器群组的支撑才能够安装使用，并且发挥最佳的性能优势。至于多少应用需要多少台服务器的支持，也没有一个固定的搭配方式，具体还要由服务器的配置和性能来决定。我们既可以租用云服务商的服务器，也可以通过购置服务器来组建一个分布式环境。如果是自己购置服务器，最好是配合使用虚拟技术，将一台服务器虚拟出多个主机来使用是一种最好的资源搭配。下面

就以搭建一个电商平台的测试环境为例，来说明一下服务器的组建过程和方法。

现在，假设我们已经有两台物理机，每台物理机的各个配件的参数如表 12-1 所示。

表 12-1 一台物理机服务器的配置表

配 件	规 格	数 量	备 注
CPU	酷睿 8i (4 核)	2	8 个逻辑 CPU
内存	32GB	2	总共 64GB
硬盘	1TB	1	
网卡	10/1000M	1	

我们使用如表 12-2 所示的配置来设计一个虚拟主机。

表 12-2 单个虚拟主机配置表

配 件	配 置	数 量
CPU	单核逻辑 CPU	2
内存	8GB	1
硬盘	80GB	1
网卡	10/1000M	1 或 2

这样，一台物理机可以通过使用 VMware 等工具虚拟出大约 10 个虚拟主机出来，即相当于 10 台逻辑服务器。而每台逻辑服务器，大约可以部署 6 个微服务应用。对于我们的电商平台实例来说，使用两台物理机已经足够安排了。当然，这其中还包括数据库的集群建设、分布式文件系统的构建等基础设施所需的服务器。

12.2 安全的服务器架构设计

服务器的安全设计是服务器组建的首要任务，安全设计的第一道屏障是防火墙的构建。有关防火墙构建的最好选择就是使用硬件防火墙设备，如果没有使用硬件防火墙，至少也要安装一个软件防火墙。下面介绍一种软件防火墙的安装及其使用方法，它的使用和配置也有点类似于硬件防火墙。

12.2.1 防火墙安装及配置

软件防火墙我们将使用 Kerio WinRoute Firewall (KWF)，这是一个包含网关服务的防火墙系统，并且还提供了 VPN 服务，它与硬件防火墙 Juniper 的配备有点类似。

KWF 只能安装在 Windows 操作系统上, 所以我们使用一个虚拟机来安装 Windows 2008 操作系统。因为防火墙的主机还要当作一个网关来使用, 所以这台虚拟机必须配置两个虚拟网卡, 一个作为外网(在测试环境中, 我们可以将局域网作为外网来看待)通信, 另一个作为内网的网关。

假设我们正在使用的局域网的网段为“192.168.1.*”, 那么现在可以将一个网卡的 IP 设置为“192.168.1.30”, 并确保这个 IP 未被其他机器使用。配置完成如图 12-1 所示。

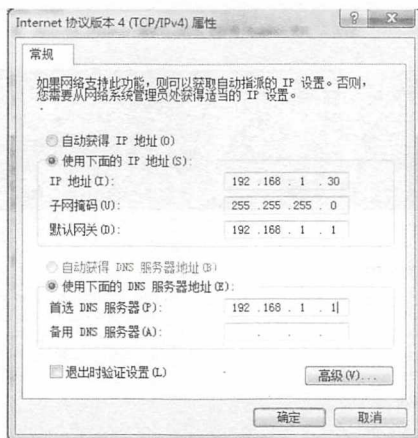


图 12-1 外网网卡配置

我们可以将内网的网段设定为“10.10.10.*”, 所以作为网关使用的另一个网卡的 IP 可以设置为“10.10.10.1”, 如图 12-2 所示。

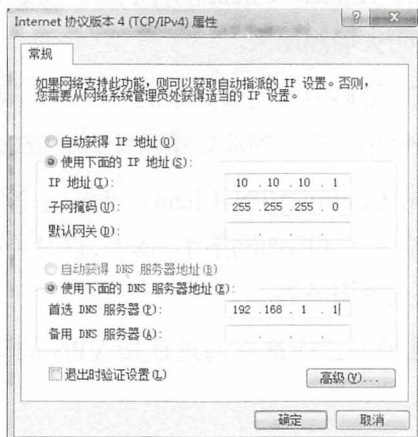


图 12-2 内网网关配置

需要注意的是，这个网卡不再需要设置网关。

现在可以安装 KWF。KWF 的安装包可以从如下网址下载：

<http://download.kerio.com/archive/>

打开链接后在产品列表上选择 **Kerio WinRoute Firewall**，并选择一个合适的版本，就可打开一个下载页面，如图 12-3 所示。这里是在 Windows 2008 上安装，所以可以选择 Windows 64 bit 的安装包下载。



图 12-3 KWF 下载

安装时最好是在 VMware vSphere Client 的控制台上安装，因为如果使用远程桌面安装的话，安装完成后将会被防火墙踢出来。

安装完成之后可以在流量策略中使用引导生成一个默认规则，这个时候也可以创建一个远程桌面访问的规则以备使用。默认的流量策略配置完成之后如图 12-4 所示。

现在，再安装一个 Kerio Control VPN Client，通过这个客户端来登录防火墙的 VPN 服务，就可以直接管理防火墙下面的内部网络了。例如，这时可以使用内网 IP “10.10.10.1”，直接使用远程桌面来登录防火墙服务器。

后面的服务器管理和维护等工作都将通过连接 VPN 的基础上，直接在内网上进行操作。

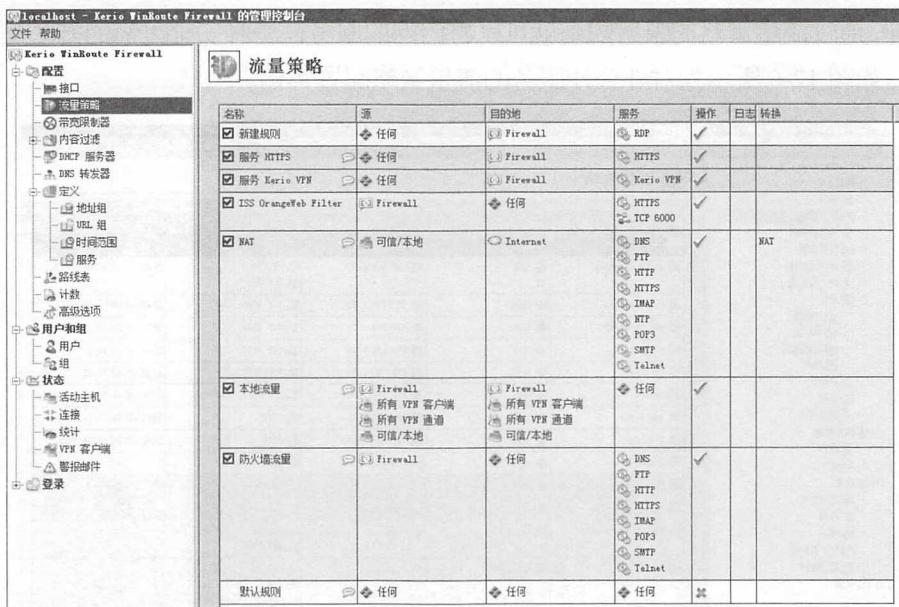


图 12-4 KWF 默认流量策略配置

在内网环境中，如果要对外网公开服务，可以通过如下方式进行配置。

第一，在防火墙的外网网卡上，增加外网的 IP，如图 12-5 所示。

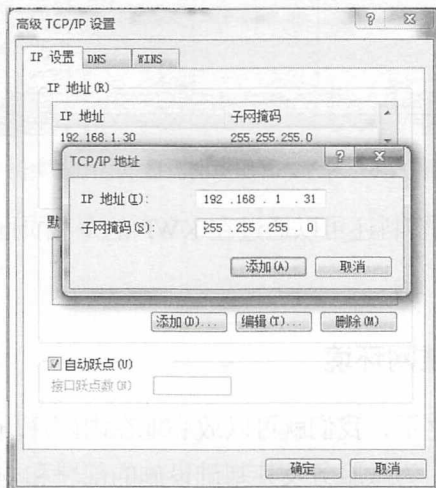


图 12-5 增加外网 IP

第二，在防火墙的流量策略中新建一个规则，将外网的访问映射到内网的服务之中。

这样,通过流量策略的映射规则就可以将内网的服务以一种安全可靠的方法提供给外网使用。如图 12-6 所示为对外公开服务的流量策略配置实例。

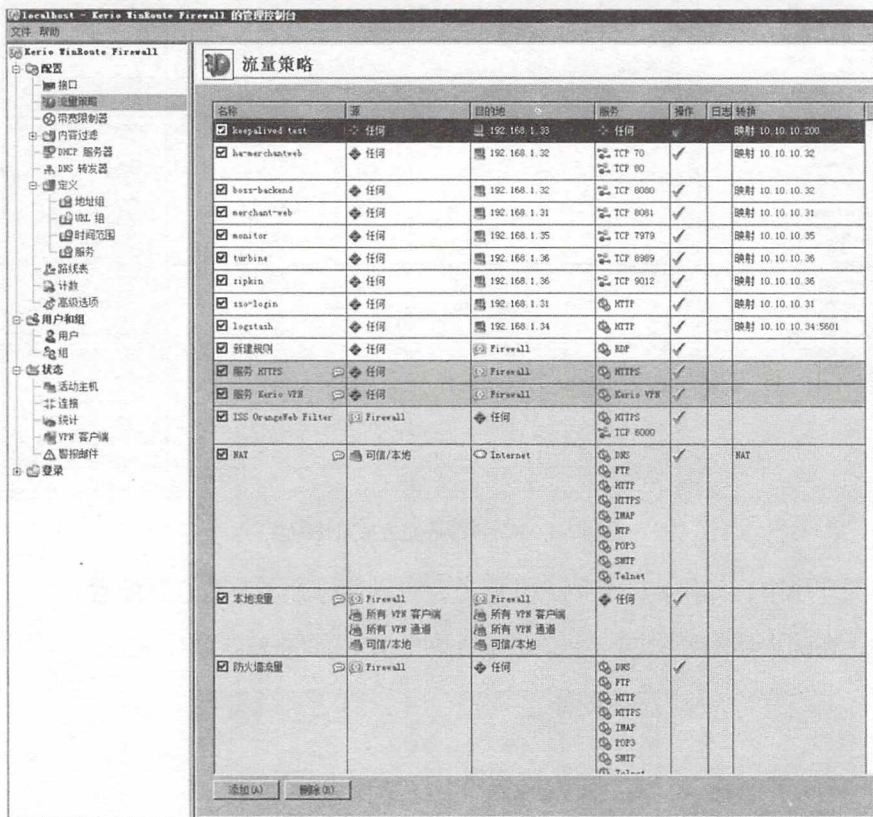


图 12-6 对外公开服务的流量策略配置实例

有关 KWF 的更多详细资料还可以通过在 KWF 的下载页面下载各种相关的文档进行查阅。

12.2.2 建立安全的局域网环境

在防火墙的安全管理之下,我们就可以放心地在内网环境中搭建服务器,从而建立起一个安全的局域网环境,为微服务及其基础设施的部署和构建提供一个安全可靠并且可持续扩展的分布式环境。

除了防火墙我们使用 Windows 操作系统之外,其他虚拟机都将使用 Linux 操作系统,

我们将使用开源的 CentOS 7.0 或以上的版本来安装。这些操作系统的 IP 都将使用内网网址来配置，如果需要对外提供服务，就在防火墙的流量策略中建立相关的规则进行访问。

设置好防火墙并规划了服务器之后，我们可以画出电商平台的网络拓扑结构图，如图 12-7 所示。

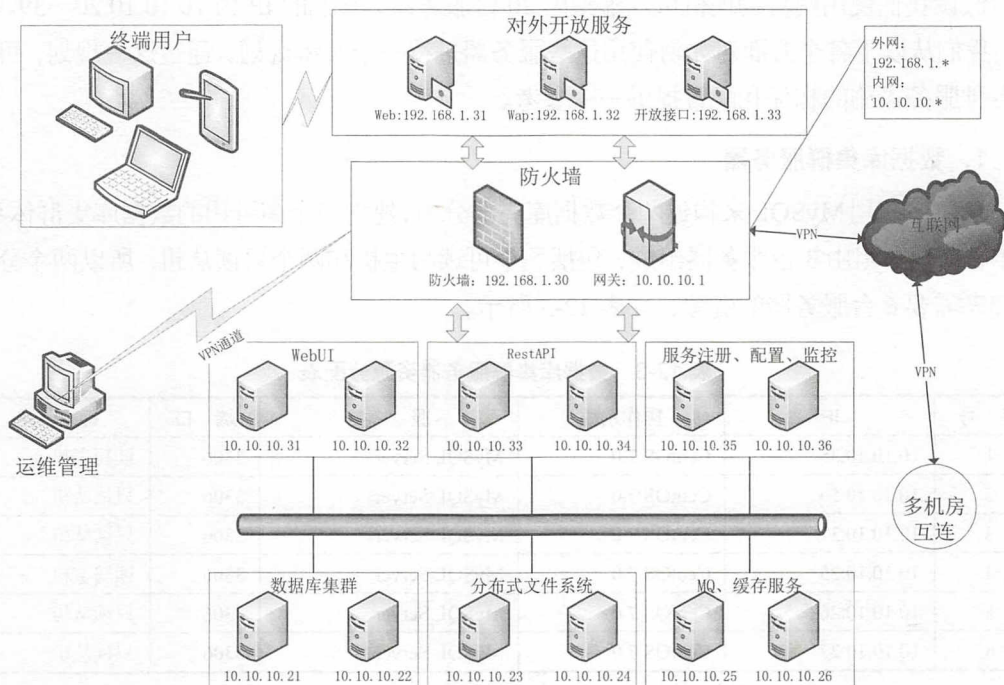


图 12-7 电商平台的网络拓扑结构图

在这个网络拓扑结构中，电商平台的各种服务器及其使用的数据库、文件系统和 MQ 服务器等都处在防火墙的安全防护之中，并使用内网 IP 与防火墙的网关进行连接，运维管理只能通过 VPN 通道来管理服务器。电商平台通过防火墙的流量策略设置对外提供有限的服务。后面的服务器安装和配置都将根据这个网络拓扑结构进行组建。

另外，在这个网络结构设计中，除了在一个防火墙中建立起来的局域网，还可以通过将防火墙与其他地区或不同机房的局域网连接起来，组成一个更大的网络体系。

12.3 服务器资源分配

对于服务器资源的分配，必须有一个合理的规划才能让有限的资源得到充分的使用。另外，合理配置各种服务的端口，还可以对服务器资源进行交叉使用。

假设我们使用两台物理机可以虚拟出 20 台服务器，它们的 IP 为 10.10.10.20~39。现在，我们从以下各个方面对如何使用这些服务器进行一个统筹规划，通过这一规划，可以为各种服务设施的构建和配置提供一个参考。

1. 数据库集群服务器

我们将使用 MySQL 来构建两个数据库集群分组，建立一个高可用的数据库集群体系，其中，每个分组由 3 台服务器组成，包括一个可读写主机和两个只读从机，所以两个分组加起来需要 6 台服务器的资源，如表 12-3 所示。

表 12-3 数据库集群服务器资源分配表

序 号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.35	CentOS 7.0	MySQL Server	3306	读写主机
2	10.10.10.36	CentOS 7.0	MySQL Server	3306	只读从机
3	10.10.10.37	CentOS 7.0	MySQL Server	3306	只读从机
4	10.10.10.25	CentOS 7.0	MySQL Server	3306	读写主机
5	10.10.10.26	CentOS 7.0	MySQL Server	3306	只读从机
6	10.10.10.27	CentOS 7.0	MySQL Server	3306	只读从机

2. 分布式文件系统服务器

分布式文件系统将使用开源的 FastDFS 来搭建，假如配置两个跟踪服务器（Tracker Server）和两个存储节点（Tracker Storage），则总共要占用 4 台服务器的资源，如表 12-4 所示。

表 12-4 分布文件系统服务器分配表

序 号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.22	CentOS 7.0	Tracker Server	22122	文件服务
2	10.10.10.23	CentOS 7.0	Tracker Storage	23000	存储节点
3	10.10.10.32	CentOS 7.0	Tracker Server	22122	文件服务
4	10.10.10.33	CentOS 7.0	Tracker Storage	23000	存储节点

3. 微服务治理体系服务器

微服务治理环境假如配置两个注册管理中心、一个配置管理中心，而服务监控、聚合服务监控、服务跟踪管理和日志服务器等各安装一个服务，这些服务的安装通过合理搭配，大约需要占用 3 台服务器的资源，如表 12-5 所示。

表 12-5 微服务治理服务器资源分配表

序 号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.31	CentOS 7.0	Eureka	8761	注册中心之一
			Config	8888	配置服务器
			Hystrix	7979	监控服务器
2	10.10.10.21	CentOS 7.0	Eureka	8761	注册中心之二
			Turbine	8989, 8990	聚合监控
			Zipkin	9987	跟踪服务
3	10.10.10.32	CentOS 7.0	Logback	5000, 5601	日志分析平台

4. 数据库代理使用的服务器

数据库代理服务将使用 OneProxy，它可以对 MySQL 的主从服务器实现高可用的读写分离设计，并可根据需要大容量数据进行分区表设计。如果再在 OneProxy 上面进行双机热备设计，则至少占用 4 台服务器的资源，如表 12-6 所示。

表 12-6 双机热备 OneProxy 代理服务器资源分配表

序号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.24	CentOS 7.0	OneProxy	3306	双机热备
2	10.10.10.34	CentOS 7.0	OneProxy	3306	双机热备

5. 消息服务使用的服务器

消息服务我们将使用 RabbitMQ 服务器，可以预留两台服务器的资源，以后根据规模发展可以考虑建立一个集群体系，其服务器资源分配如表 12-7 所示。

表 12-7 RabbitMQ 服务器资源分配表

序号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.29	CentOS 7.0	RabbitMQ	15672, 5672	消息服务器
2	10.10.10.30	CentOS 7.0	RabbitMQ	15672, 5672	消息服务器

6. 缓存服务器

缓存将使用 NoSQL 数据库 Redis 来实现，所以也可以准备两台服务器资源，以后根据需要同样可以建立集群体系，如表 12-8 所示。

表 12-8 Redis 服务器资源分配表

序号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.29	CentOS 7.0	Redis	6379	缓存
2	10.10.10.39	CentOS 7.0	Redis	6379	缓存

7. Git 代码仓库服务器

Git 代码仓库主要为配置管理中心提供安全的文件管理服务，为了更好管理可以安装一个 GitLab，只需占用一台服务器的资源，如表 12-9 所示。

表 12-9 GitLab 服务器资源分配表

序号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.20	CentOS 7.0	GitLab	80	代码仓库

8. 电商平台微服务应用服务器

电商平台占用的服务器资源较多，在资源分配中我们先考虑单实例部署的情况，按照现在服务器的配置来计算，一台服务器大约可以运行 6 个微服务应用，所以整个电商平台大约需要 10 台服务器，其资源分配如表 12-10 所示。在生产环境中，随着业务规模的发展，将会进行多实例、多副本的发布，这将需要有更多的服务器资源。

表 12-10 电商平台微服务应用服务器资源分配表

序号	IP	操作系统	服 务	端 口	备 注
1	10.10.10.22	CentOS 7.0	catalog-restapi	9091	类目服务接口
			catalog-web	8091	类目 PC 应用
			catalog-wap	7091	类目移动应用
2	10.10.10.23	CentOS 7.0	goods-restapi	9092	商品服务接口
			goods-web	8092	商品 PC 应用
			goods-wap	7092	商品移动应用
3	10.10.10.24	CentOS 7.0	order-restapi	9093	订单服务接口
			order-web	8093	订单 PC 应用
			order-wap	7093	订单移动应用

续表

序号	IP	操作系统	服 务	端 口	备 注
4	10.10.10.25	CentOS 7.0	customer-restapi	9094	顾客服务接口
			customer-web	8094	顾客 PC 应用
			customer-wap	7094	顾客移动应用
5	10.10.10.26	CentOS 7.0	member-restapi	9095	会员服务接口
			member-web	8095	会员 PC 应用
			member-wap	7095	会员移动应用
6	10.10.10.27	CentOS 7.0	payment-restapi	9096	支付服务接口
			payment-web	8096	支付 PC 应用
7	10.10.10.32	CentOS 7.0	track-restapi	9097	浏览记录接口
			track-web	7097	浏览记录应用
8	10.10.10.33	CentOS 7.0	comment-restapi	9098	评价服务接口
			comment-wap	7098	评价移动应用
9	10.10.10.34	CentOS 7.0	shopcart-restapi	9083	购物车服务接口
			shopcart-wap	7083	购物车移动应用
10	10.10.10.35	CentOS 7.0	logistics-restapi	9084	物流服务接口
			logistics-web	8084	物流 PC 应用
			logistics-wap	7084	物流移动应用
11	10.10.10.36	CentOS 7.0	merchant-restapi	9081	商家服务接口
			merchant-web	8081	商家 PC 应用
			merchant-sso	8000	单点登录服务
			manage-web	8099	平台管理应用

12.4 CentOS 安装

服务器使用的 Linux 操作系统都使用了 CentOS 来进行安装, CentOS 是一个开源的 Linux 发行版, 具有很好的稳定性和更多的可扩展性。为了能够正常使用 Docker, 我们将使用 CentOS 7 及以上的版本。

CentOS 的安装包可以通过如下网址进行下载:

<http://www.centoscn.com/>

这里的安装以 CentOS 7.1 为例进行说明, 安装时使用最小安装选项来安装, 下面主要介绍安装过程中的几个重要设置。

12.4.1 IP 地址设置

在安装配置中可以先进行网络设置，使用网段“10.10.10.*”来设置 IP 地址，网关和 DNS 都指定为“10.10.10.1”，即防火墙服务器配置的网关。

如果未配置好 IP，也可以在安装完成之后，使用下列指令来设置：

```
# vi /etc/sysconfig/network-scripts/ifcfg-ens192
```

即编辑网卡配置，主要设置类似于如下所示的几个参数：

```
ONBOOT=yes
IPADDR=10.10.10.31
NETMASK=255.255.255.0
NETWAY=10.10.10.1
DNS1=10.10.10.1
```

要让更改的 IP 立即生效，可以使用下列指令重启网络：

```
# service network restart
```

使用下列指令可以查看 IP 地址信息：

```
# ip addr
```

12.4.2 安全设置

因为我们的服务器已经处在防火墙安全保护的环境之中，所以为了省略后面安装其他服务时的防火墙配置，可以将 Linux 的安全选项禁用掉。

关闭强制访问控制功能使用如下指令编辑系统配置：

```
# vi /etc/selinux/config
```

将其中的“SELINUX=enforcing”修改为“SELINUX=disabled”。

禁用防火墙使用如下指令：

```
# systemctl stop firewalld.service #停止 firewall
# systemctl disable firewalld.service #禁止 firewall 开机启动
```

12.4.3 语言配置

为了在终端上能正确显示中文信息，使用如下指令编辑语言配置：

```
# vi /etc/locale.conf
```

将内容更改为如下所示：


```
LANG="zh_CN.UTF-8"  
SUPPORTED="zh_CN.UTF-8:zh_CN:zh"  
SYSFONT="latarcyrheb-sun16"
```

保存退出后, 使用如下指令让配置立即生效:

```
# source /etc/locale.conf
```

12.4.4 时间同步配置

安装一个时间同步服务器, 可以校准服务器的时间。使用如下指令可以安装一个时间同步服务:

```
# yum -y install ntp
```

安装完成后, 使用如下两条指令启动时间同步服务, 并将其设置为开机启动:

```
# systemctl enable ntpd  
# systemctl start ntpd
```

12.5 Docker 和 docker-compose 安装

Docker 是一个优秀的容器引擎。使用 Docker 可以为应用系统创建一个可移植的容器, 容器运行于宿主系统上, 其功能相当于一个虚拟主机, 但是与虚拟主机相比, 它具有更多的优势。Docker 容器占用资源少, 构建非常灵活和方便, 可以非常快速地启动和关闭。电商平台的微服务应用都将使用 Docker 来发布。

我们开发的微服务已经自包含了 Tomcat 中间件, 打包后的 Jar 文件可以使用如下所示的 Java 指令直接运行:

```
java -jar *.jar
```

所以, 使用 Docker 来部署我们开发的微服务是非常简单的, 只要使用类似于上面所示的指令就可以在 Docker 中运行 Jar 包。

使用 Docker 来部署微服务还可以利用更多的服务器资源, 而且设置简单、操作方便, 而服务的更新和运行将更加快速和高效。

下面介绍一下 Docker 及其工具的安装, 并对 Docker 的使用也进行了一些简单的说明, 以加深对 Docker 的认识和理解。

12.5.1 Docker 安装及使用

在 CentOS 7 中安装 Docker 也很简单，可以按以下步骤进行。

首先，使用如下指令更新一下安装环境：

```
# yum update
```

其次，编辑如下命令以配置 Docker 的安装源：

```
# tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

然后，使用如下指令开始安装：

```
# yum install docker-engine
```

安装需要一定的时间，将会通过网络下载一些安装文件。

安装完成后可以使用如下指令启动 Docker：

```
# service docker start
```

使用如下指令检查版本：

```
# docker --version
```

如果启动正常，将输出类似于如下所示的版本信息：

```
Docker version 17.09.0-ce, build afdb6d4
```

查看详细的版本信息，还可以使用如下指令：

```
# docker version
```

最后，使用如下指令可以将 Docker 设置为开机启动：

```
# systemctl enable docker
```

使用 Docker 运行一个服务时，必须首先创建这个服务的镜像，然后使用这个镜像来创建容器并运行。在同一宿主中，一个镜像可以创建多个容器副本。所以，当需要更新一个服务时，必须删除所有的容器，然后更新其相关的镜像。

为了加深对 Docker 的理解，下面我们以部署一个注册中心为例进行演示说明。

1. 在“base-microservice”工程中执行打包

打包成功后，在“base-eureka”模块中将生成一个发布包：base-eureka-1.0-SNAPSHOT.jar。

在命令行窗口或终端上使用如下指令运行发布包，以确认应用能够正常运行：

```
java -jar base-eureka-1.0-SNAPSHOT.jar
```

2. 在服务器上生成镜像

在服务器上创建一个工作目录“/base/uereka”，将上面的发布包上传到这个目录中，并在这个目录中创建一个“Dockerfile”文件，这个文件是生成镜像的脚本，文件内容如下所示：

```
FROM java:8
VOLUME /tmp
ADD base-eureka-1.0-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
    && echo 'Asia/Shanghai' >/etc/timezone
EXPOSE 8761
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

这个脚本表示使用 Java 8 镜像，将注册中心的发布包生成一个由 Java 运行的镜像，其中使用“EXPOSE”指定了运行应用时设定的端口号，另外，设定“Shanghai”时区的目的是让应用运行时输出的日志能够显示正确的时间。

其他应用的镜像生成脚本都可以参照这个脚本来创建，只要修改相关的发布包文件名和端口号即可。

使用“build”指令来生成镜像，如下所示是执行指令及其运行的结果：

```
docker build -t eureka .
Sending build context to Docker daemon 39.82MB
Step 1/6 : FROM java:8
----> d23bdf5b1b1b
Step 2/6 : VOLUME /tmp
----> Running in d1bab95a45cb
----> 2859a98464bc
Removing intermediate container d1bab95a45cb
Step 3/6 : ADD base-eureka-1.0-SNAPSHOT.jar app.jar
----> 418d9ca504e8
Step 4/6 : RUN bash -c 'touch /app.jar'
```

```

---> Running in 63be983f3386
---> bffff9a32aa9d
Removing intermediate container 63be983f3386
Step 5/6 : EXPOSE 8761
---> Running in 6e88bf965796
---> fd10ddd6ff1f
Removing intermediate container 6e88bf965796
Step 6/6 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar
/app.jar
---> Running in 435836ac1924
---> e2b9211094c5
Removing intermediate container 435836ac1924
Successfully built e2b9211094c5

```

其中，如果你的机器上不存在 Java 镜像的话，将会从镜像服务器上寻找并拉取下来。

现在，可以使用“images”指令查看已经生成的镜像，如下所示为执行指令及其输出的结果：

```

docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
eureka	latest	e2b9211094c5	6 minutes ago	723MB
java	8	d23bdf5b1b1b	10 months ago	643MB

其中，“eureka”即是我们生成的镜像，因为没有指定版本号，它的 TAG 默认设置为“latest”。

3. 运行容器

通过如下指令可以使用镜像生成并运行一个容器：

```
docker run --name eureka_1 -d -p 8761:8761 eureka
```

其中，“name”参数指定的就是容器的名字，端口设定中冒号前面设置宿主机器的端口，冒号后面设置容器的端口，即“宿主端口:容器端口”，如果没有冒号即只有一个端口号，那么就是设置容器的端口，最后一个“eureka”是所使用的镜像的名字。

现在使用“ps”指令可以查看正在运行的容器，如下所示是执行指令及其输出结果：

```

docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
dc68cad98920	eureka	"java -Djava.secur..."	9 seconds ago	Up 8 seconds

0.0.0.0:8761->8761/tcp eureka_1

使用如下指令可以查看运行的容器中控台的输出日志：


```
docker logs -f eureka_1
```

4. 删除容器和镜像

一个应用的重启和更新及其相关容器和镜像的管理等操作,可以使用如下所示的一些 Docker 指令来完成。

启动容器:

```
docker start eureka_1
```

停止容器:

```
docker stop eureka_1
```

删除容器:

```
docker rm eureka_1
```

删除镜像:

```
docker rmi eureka
```

对于 Docker 的使用,我们将主要使用一个更加高效的工具来进行管理和操作,所以很少直接使用到 Docker 指令来部署一个应用。

需要了解更多有关 Docker 的信息,可以通过如下链接访问其官方网站:

```
https://docs.docker.com/
```

12.5.2 docker-compose 安装及使用

docker-compose 是一个通过编排脚本来使用 Docker 引擎的工具组件,使用这一工具将使我们不用记住那么多命令和配置参数,可以更加方便和快速地部署和更新一个应用的发布。

使用如下指令可以将已经编译的 docker-compose 工具下载到本地系统中:

```
curl -L https://github.com/docker/compose/releases/download/1.16.0-rc2/  
docker-compose -o /usr/local/bin/docker-compose
```

其中,“1.16.0-rc2”为版本号,可以先从 GitHub 中查看 docker-compose 的最新版,然后更改上面下载指令的版本号,下载最新的版本。

下载完成后执行如下指令,更改 docker-compose 工具的执行权限:

```
chmod +x /usr/local/bin/docker-compose
```

使用如下指令查看 docker-compose 的版本号:

```
docker-compose version
```

如果返回类似于如下所示的信息，则表明安装完成：

```
docker-compose version 1.16.1, build 6dlac21
docker-py version: 2.5.1
CPython version: 2.7.12
OpenSSL version: OpenSSL 1.0.2j 26 Sep 2016
```

执行如下指令，将可以输出 **docker-compose** 工具完整的帮助信息：

```
# docker-compose -h
```

如下所示是输出结果，我们可以看一看各种指令的详细说明：

```
Define and run multi-container applications with Docker.
```

Usage:

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
docker-compose -h|--help
```

Options:

<code>-f, --file FILE</code>	Specify an alternate compose file (default: docker-compose.yml)
<code>-p, --project-name NAME</code>	Specify an alternate project name (default: directory name)
<code>--verbose</code>	Show more output
<code>--no-ansi</code>	Do not print ANSI control characters
<code>-v, --version</code>	Print version and exit
<code>-H, --host HOST</code>	Daemon socket to connect to
<code>--tls</code>	Use TLS; implied by <code>--tlsverify</code>
<code>--tlscacert CA_PATH</code>	Trust certs signed only by this CA
<code>--tlscert CLIENT_CERT_PATH</code>	Path to TLS certificate file
<code>--tlskey TLS_KEY_PATH</code>	Path to TLS key file
<code>--tlsverify</code>	Use TLS and verify the remote
<code>--skip-hostname-check</code>	Don't check the daemon's hostname against the name specified
	in the client certificate (for example if your docker host
	is an IP address)
<code>--project-directory PATH</code>	Specify an alternate working directory (default: the path of the Compose file)

Commands:

<code>build</code>	Build or rebuild services
<code>bundle</code>	Generate a Docker bundle from the Compose file

config	Validate and view the Compose file
create	Create services
down	Stop and remove containers, networks, images, and volumes
events	Receive real time events from containers
exec	Execute a command in a running container
help	Get help on a command
images	List images
kill	Kill containers
logs	View output from containers
pause	Pause services
port	Print the public port for a port binding
ps	List containers
pull	Pull service images
push	Push service images
restart	Restart services
rm	Remove stopped containers
run	Run a one-off command
scale	Set number of containers for a service
start	Start services
stop	Stop services
top	Display the running processes
unpause	Unpause services
up	Create and start containers
version	Show the Docker-Compose version information

这些指令中，我们经常会用到的是“build”“up”“ps”“start”“stop”“down”“logs”等指令。如果要发布一个微服务，我们只要简单地使用一个“up”指令就足够了。

另外，还可以使用“help”指令来查看每一个指令的帮助信息，例如，要查看“down”指令的使用说明，可以使用如下所示的方法：

```
docker-compose help down
```

执行指令后将输出如下所示的结果：

```
Stops containers and removes containers, networks, volumes, and images
created by `up`.
```

```
By default, the only things removed are:
```

- Containers for services defined in the Compose file
- Networks defined in the `networks` section of the Compose file
- The default network, if one is used

```
Networks and volumes defined as `external` are never removed.
```

```
Usage: down [options]
```

```
Options:
```

```
--rmi type          Remove images. Type must be one of:
                    'all': Remove all images used by any service.
                    'local': Remove only images that don't have a custom tag
                    set by the `image` field.

-v, --volumes        Remove named volumes declared in the `volumes` section
                    of the Compose file and anonymous volumes
                    attached to containers.

--remove-orphans      Remove containers for services not defined in the
                    Compose file
```

上面使用 Docker 部署注册中心的例子，如果使用 docker-compose 工具来部署，将会变得非常简单。

首先，我们在“/base”目录中创建一个“docker-compose.yml”文件，使用这一文件就可以编排部署脚本。对于这个例子来说，可以编排如下所示的脚本：

```
eureka:
  build: ./eureka
  ports:
    - "8761:8761"
```

然后，就可以使用 docker-compose 的“up”指令来部署应用了，如下所示：

```
docker-compose up -d
```

运行这个指令已经包含了镜像的创建、容器的生成和启动等操作。其中，参数“-d”表示在后台中运行。

当需要更新部署时，只需要一个“down”指令也可以完成，如下所示：

```
docker-compose down --rmi all
```

运行这个指令将停止由编排脚本所管理的所有容器，然后删除相关的容器和镜像。

所以，使用 docker-compose 工具来部署一个应用，我们只要使用一个简单的指令就可以完成所有应该做的事情。后面的微服务部署的工作，我们都将使用这一工具来完成，包括自动化部署设施的建设也离不开这一优秀的工具组件。

12.6 使用 Docker 搭建微服务治理环境

完成 Docker 的安装之后就可以构建微服务的治理环境。微服务治理主要以注册中心为基础,通过服务注册与发现的功能,对微服务的调用、实现负载均衡调度和智能路由服务,同时配合配置管理、服务监控和跟踪,对分布式环境之中的微服务实现更加高效的管理和调控。

我们将部署具有两个注册中心、相关的配置管理中心、监控中心和服务跟踪管理等所构成的一个高可用的微服务治理环境,如图 14-8 所示。

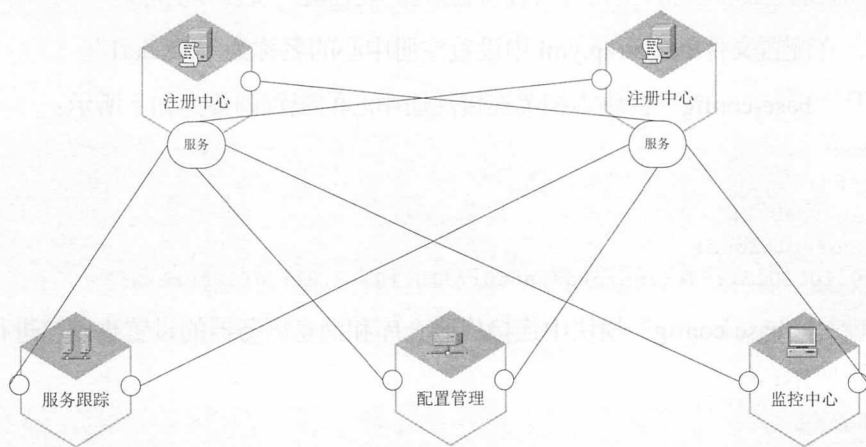


图 12-8 高可用的微服务治理环境

部署这个治理环境将使用两台服务器,这两台服务器的编号和 IP 地址如下所示:

服务器 1: 10.10.10.31

服务器 2: 10.10.10.21

其中,注册中心将分别在两台服务器各部署一个实例。下面说明部署的过程和方法。

12.6.1 服务器 1 的部署配置

在服务器 1 中将部署一个注册中心、一个配置管理中心和一个监控中心,所以我们可以创建如下所示的目录结构:

```
/base/eureka
/base/config
/base/hystrix
```

在微服务开发的过程中,“base-microservice”工程中的一些配置只是设置为方便本地调试使用,现在要发布到测试环境中,必须做一些相关调整。

首先是“base-eureka”模块中有关“eureka”的配置可调整为如下所示:

```
eureka:
  instance:
    hostname: 10.10.10.31
  client:
    registerWithEureka: true
    fetchRegistry: false
    serviceUrl:
      defaultZone: http:// 10.10.10.21:${server.port}/eureka/
```

另外,在配置文件 bootstrap.yml 中设置注册中心的名称为“eureka1”。

其次是“base-config”模块中相关连接注册中心的配置调整为如下所示:

```
eureka:
  client:
    serviceUrl:
      defaultZone:
http://10.10.10.31:8761/eureka/,http://10.10.10.21:8761/eureka/
```

还有就是“base-config”模块中连接代码仓库和消息服务器的设置也应该进行调整,完成后如下所示:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://10.10.10.20/demo/base-config-repo.git
          username: demo
          password: 12345678

  rabbitmq:
    addresses: amqp://10.10.10.29:5672
    username: alan
    password: alan
```

其中的用户名和密码视各个服务器的配置而定。

“base-hystrix”模块因为没有相关服务的连接配置,所以并不需要调整。

上面所有配置调整之后,将工程进行重新打包,并将各个模块的发布包上传到上面创

建的目录结构的相关子目录中，同时在各个子目录中创建相关的 Dockerfile。

最后，在“/base”目录下面创建一个“docker-compose.yml”文件，并编写如下所示的内容：

```
eureka:
  build: ./eureka
  ports:
    - "8761:8761"
config:
  build: ./config
  ports:
    - "8888:8888"
  links:
    - eureka
hystrix:
  build: ./hystrix
  ports:
    - "7979:7979"
```

这样，就完成了服务器 1 的部署准备工作，当相关的 RibbitMQ 和 GitLab 安装完成之后，就可以使用 Docker-compose 工具的“up”指令和“down”指令进行部署管理了。

12.6.2 服务器 2 的部署配置

在服务器 2 中，部署一个注册中心、一个聚合服务监控中心和一个跟踪服务器，所以也可以参照上面的方法一起统一进行部署。

首先，根据各个服务的名字，在这个服务器中创建如下所示的目录结构：

```
/base/eureka
/base/turbin
/base/zipkin
```

其次，各个应用中需要调整的配置包括如下几个方面。

1. 注册中心的配置修改

注册中心的连接配置只将 IP 进行相关调整，如下所示：

```
eureka:
  instance:
    hostname: 10.10.10.21
  client:
    registerWithEureka: true
```



```
fetchRegistry: false
serviceUrl:
  defaultZone: http:// 10.10.10.31:${server.port}/eureka/
```

另外，在配置文件 `bootstrap.yml` 中设置注册中心的名称为“eureka2”。

这样，两个注册中心相互进行连接，将可共享注册实例信息。

2. 聚合监控中心的配置修改

将连接注册中心的配置改成如下所示：

```
eureka:
  client:
    serviceUrl:
      defaultZone:
http://10.10.10.31:8761/eureka/,http://10.10.10.21:8761/eureka/
```

其中，有关所监控的服务列表配置可以在配置管理中心进行配置和更新。

而连接配置管理中心和消息服务器的配置，做出如下所示的修改：

```
spring:
  cloud:
    config:
      uri: http://10.10.10.31:8888

  rabbitmq:
    addresses: amqp://10.10.10.29:5672
    username: alan
    password: alan
```

3. 跟踪服务的配置修改

有关连接注册中心的配置可以参照上面聚合监控服务的配置进行修改，而连接数据库的配置可以将原来的“localhost”更改为“10.10.10.24”，即安装 Onepoxy 代理服务器的 IP，用户名和密码可以依据代理服务器的设置进行修改。

上面配置修改完成之后就可以将工程重新打包，将相关应用的发行包上传到上面创建的名字相同的子目录之中，然后创建相关的 `Dockerfile`。

最后，在“base”目录中创建一个“`docker-compose.yml`”文件，并在文件中编写如下所示的内容：

```
eureka:
  build: ./eureka
  ports:
```




```

- "8761:8761"
turbine:
  build: ./turbine
  ports:
    - "8989:8989"
    - "8990:8990"
  links:
    - eureka
zipkin:
  build: ./zipkin
  ports:
    - "9987:9987"
  links:
    - eureka

```

这样，一个以高可用的注册中心为基础的微服务治理环境的部署配置就已经准备就绪，等待相关的服务器安装完成之后，就可以启动运行。

12.7 使用 Docker 部署日志分析平台

微服务应用生成的日志，我们将使用一个统一的日志分析平台来管理，这将给日志的查询和使用提供极大的方便。

日志分析平台 ELK 由 3 个服务组成，分别是 Elasticsearch、Logstash 和 Kibana，其中：

- Elasticsearch 是一个分布式搜索分析引擎，负责日志存储并提供搜索功能。
- Logstash 能提供数据收集、加工和传输管道的服务，负责日志收集。
- Kibana 是一个数据可视化平台，并可以将数据分析结果转化为图表等形式，即提供了 Web 查询的操作界面。

因为日志分析平台 ELK 中的 3 个服务都是开源的，并且已经发布到镜像仓库中，所以我们将通过 Docker 使用其提供的镜像来进行部署和安装。

首先，在服务器上创建一个目录，如下所示：

```
mkdir /logstash
```

进入这个目录之后，使用如下所示的指令创建一个配置文件：

```
vi logstash.conf
```

文件的内容如下所示：

```
input {
```



```
tcp {
  port => 5000
  codec => json
}
udp {
  port => 5000
  codec => json
}

output {
  elasticsearch { hosts => [ "elasticsearch:9200" ] }
}
```

然后，使用如下指令创建一个编排脚本的文件：

```
vi docker-compose.yml
```

在文件中编写如下所示内容：

```
logstash:
  image: logstash:5.4.0
  volumes:
    - ./logstash.conf:/etc/logstash.conf
  ports:
    - "5000:5000/tcp"
    - "5000:5000/udp"
  links:
    - elasticsearch
  command:
    -f /etc/logstash.conf

elasticsearch:
  image: elasticsearch:5.4.0

kibana:
  image: kibana:5.4.0
  links:
    - elasticsearch
  ports:
    - "5601:5601"
```

其中，3 个服务的版本号必须统一。然后，使用如下所示的指令来启动服务：

```
docker-compose up -d
```

第一次启动时需要一定的时间，因为需要从镜像仓库中拉取相关的镜像。



启动成功之后就可以使用日志分析平台了。以后对日志分析平台的管理，可以通过使用 `docker-compose` 工具中的“start”“stop”等指令来执行平台的启动或关闭等操作。

在应用工程中，要使用日志分析平台的日志收集功能，可以通过日志配置文件 `logback.xml` 进行配置，如下所示是一个完整的日志配置文件的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property name="LOG_HOME" value="/logs" />
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder charset="UTF-8">
      <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}
- %msg%n</pattern>
    </encoder>
  </appender>
  <appender name="stash" class="net.logstash.logback.appender.Logstash
TcpSocketAppender">
    <destination>10.10.10.32:5000</destination>
    <encoder charset="UTF-8" class="net.logstash.logback.encoder.
LogstashEncoder" />
  </appender>

  <appender name="async" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="stash" />
  </appender>

  <!-- show parameters for hibernate sql 专为 Hibernate 定制 -->
  <logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="TRACE" />
  <logger name="org.hibernate.type.descriptor.sql.BasicExtractor" level="DEBUG" />
  <logger name="org.hibernate.SQL" level="DEBUG" />
  <logger name="org.hibernate.engine.QueryParameters" level="DEBUG" />
  <logger name="org.hibernate.engine.query.HQLQueryPlan" level="DEBUG" />

  <!-- 设置日志级别 -->
  <root level="info">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="stash" />
  </root>
</configuration>
```

其中，通过“10.10.10.32:5000”设置了日志收集服务器的 IP 和端口号。



12.8 使用 Docker 部署微服务应用

上面介绍的是使用 Docker 进行应用和服务部署的一些实例，一般在部署完成之后就很少进行修改了，除了要做扩展配置之外，所以可以由人工部署完成。而对于一个平台来说，将会有很多微服务应用，例如，我们简简单单设计的电商平台就已经有几十个应用了，而且应用的更新和迭代将会很频繁，这时候再依靠人工来部署将会是很不现实的。电商平台的微服务应用的部署将主要依赖于自动化设施来完成。在后面的章节中将介绍如何使用自动构建工具 Jenkins 实现自动化持续交付的功能。但是，不管是人工部署，还是自动化部署，其中的主要角色还是由 Docker 来担当的，所以，都将同样使用 Dockerfile 来设计镜像生成脚本，并且使用 docker-compose 工具来编排部署脚本。

使用 Docker 进行微服务部署时，还必须注意服务注册中一个 IP 配置的问题。

我们在前面章节的微服务开发实践中，为了方便调试，应用的实例注册一般都使用如下所示的设置：

```
eureka:
  instance:
    preferIpAddress: true
```

即使用本地的 IP 地址来注册应用实例。如果通过 Docker 来部署应用再使用这个配置，就有可能导致应用不能正常访问。因为在 Docker 容器中的本地 IP 地址只是容器的 IP，并不是宿主的 IP。所以在使用 Docker 部署应用时，必须将上面的配置更改成宿主服务器的 IP。例如，改成类似于如下所示的形式：

```
eureka:
  instance:
    hostname: 10.10.10.31
```

即用“hostname”来代替“preferIpAddress”，并设定为容器宿主服务器的 IP。

为了方便大家使用，上面有关 Docker 的一些脚本设计，可以从如下链接获得：

<https://gitee.com/chenshaojian/SpringCloud>

12.9 小结

本章介绍了使用虚拟机技术来组建服务器的方法，并使用防火墙和网关技术构建了一个安全可靠的分布式环境。然后，以电商平台为例说明了如何对服务器的资源进行统

筹规划，并充分利用分布式的网络环境来实施各种高可用的服务器架构设计。同时，本章还介绍了 Docker 及其相关工具的安装和应用部署的高级用法。在各种高可用的服务器架构设计中，这里只介绍了如何使用 Docker 来构建高可用的微服务治理环境，同时也实现了日志分析平台的部署和安装。

接下来，在后面的章节中将继续介绍数据库集群、分布式文件系统等高可用的服务器架构设计的组建及其实现方法。

数据库集群设计与高可用 读写分离实施

我们所设计的每个微服务应用都能适应高并发的调用，所以它所连接的数据库也必须具有这种高性能的特性，这样才能组成一个高性能的有机整体。虽然经过微服务的细分原则，数据库管理系统也跟着细分和缩小，其性能也相应得到了一定程度的提升。但是，对于一个大规模高流量的应用平台来说，这种提升的程度是远远不够的。因此，高可用和高性能的数据库架构设计，在某种程度上来说，也是属于微服务架构的设计范围。

在 MySQL 的集群设计中，我们将首先使用主从同步设计来构建数据库集群。然后，将这种集群再以分组的形式通过主主同步来实现高可用设计。而对数据库的访问，将使用 OneProxy 数据库代理中间件来实现读写分离设计。最后，对 OneProxy 的调用还可以实现双机热备设计，并使用一个虚拟的广播 IP 即 VIP 的形式对外提供服务。这个高可用的数据库集群的架构设计如图 13-1 所示。

其中，最主要的组成部分就是数据库的集群及其分组的设计。这种集群分组可以根据应用平台规模化发展的情况进行持续扩展。在下面实施的过程中，我们将建立两个集群分组，其中每个分组分别由一个主机和两个从机所组成。

需要指出一点的是，不管数据库的集群由多少分组组成，这种读写分离的高可用架构设计，对于一个微服务应用来说是完全透明的，微服务调用数据库的方式还是像以前一样

配置一个数据源进行访问,不同的是只需要将相应的连接地址改成这种高可用架构所提供的 VIP 即可。

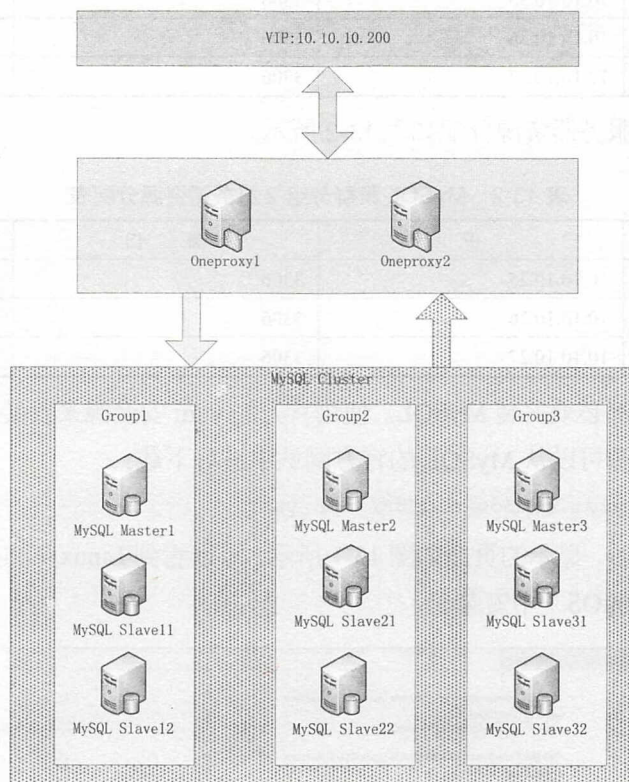


图 13-1 高可用数据库集群读写分离架构设计图

下面就从数据库安装开始,按步骤来说明怎么在分布式环境中实施这个高可用的架构设计。

13.1 MySQL 安装

我们将使用 6 台服务器来创建两个 MySQL 集群分组,其中,第一个分组的服务器资源分配如表 13-1 所示。

表 13-1 MySQL 集群分组 1 服务器资源分配表

序 号	IP	端 口	用 途
1	10.10.10.35	3306	master
2	10.10.10.36	3306	slave
3	10.10.10.37	3306	slave

第二个分组的服务器资源分配如表 13-2 所示。

表 13-2 MySQL 集群分组 2 服务器资源分配表

序 号	IP	端 口	用 途
1	10.10.10.25	3306	master
2	10.10.10.26	3306	slave
3	10.10.10.27	3306	slave

这 6 台服务器都必须安装 MySQL。使用官方的 yum 安装源来安装，可以使用最新的稳定版。yum 安装源可以从 MySQL 的官方网站中进行下载：

<https://dev.mysql.com/downloads/repo/yum/>

打开上面链接后，显示的页面如图 13-1 所示，可以看到 Linux 7 的最新版本为“5.7”，这个版本适合在 CentOS 7 中安装。

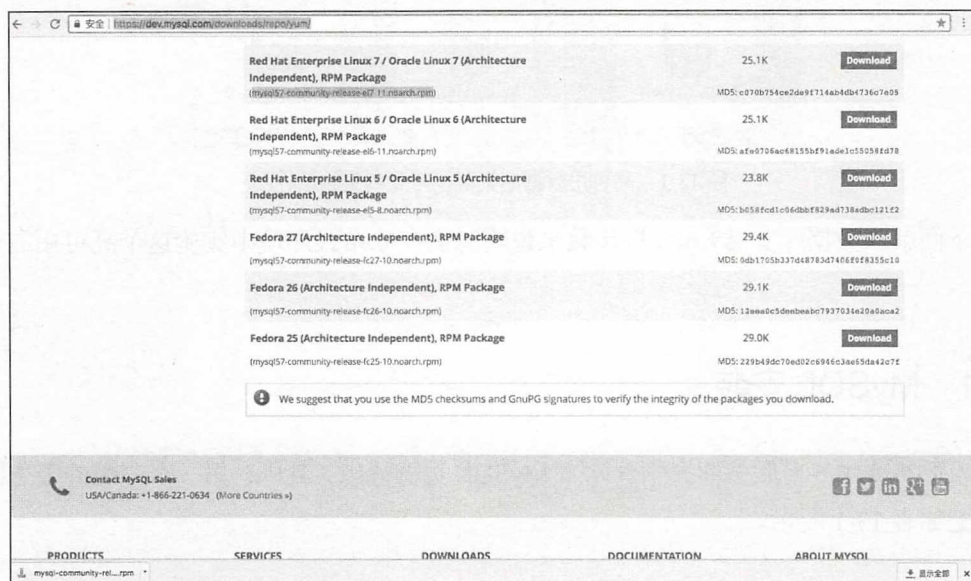


图 13-1 MySQL 官方下载网站

安装源可以从如图 13-1 所示的网站进行下载，也可以直接使用如下所示的指令在服务器中下载：

```
wget http://dev.mysql.com/get/mysql57-community-release-el7-11.noarch.rpm
```

如果服务器上没有安装“wget”，可以先使用如下指令进行安装：

```
yum install wget
```

下载完成后，在服务器本地上进行安装：

```
yum localinstall mysql57-community-release-el7-11.noarch.rpm
```

安装完成后，可以使用如下指令查看最新的稳定版本信息：

```
yum repolist all | grep mysql
```

然后，使用如下指令安装 MySQL 社区版服务器：

```
yum install mysql-community-server
```

安装完成后，可以使用如下指令将 MySQL 设置为开机启动：

```
systemctl enable mysqld
```

再使用如下指令启动 MySQL Server：

```
systemctl start mysqld
```

启动成功后，可以使用如下指令登录 MySQL 服务器：

```
mysql -u root -p
```

首次登录使用如下指令进行安全设置：

```
mysql> mysql_secure_installation;
```

安全设置可以根据提示进行设定，最后为 root 用户设置一个密码。完成设置之后，重启一下服务器，即可完成 MySQL 的安装。

13.2 主从同步设置

MySQL 的主从同步设置是将主机设定为可读写的服务器，将从机设定为只读的服务器，从机的数据将从主机中进行同步。

下面以设置“10.10.10.35”（master）与“10.10.10.36”（slave）的主从同步为例进行说明。

1. 主机设置

使用如下指令修改服务器名称：

```
vi /etc/hostname
```

将文件内容改为“mysql-35”。

使用如下指令修改数据库服务器的配置：

```
vi /etc/my.cnf
```

在[mysqld]下面增加以下配置项：

```
[mysqld]
# 服务器的 ID，必须唯一
server_id=35
# 复制过滤：不需要备份的数据库
binlog-ignore-db=mysql
# 二进制日志名字
log-bin=demo-mysql-bin
# 为每个 session 分配的内存，在事务过程中用来存储二进制日志的缓存
binlog_cache_size=1M
# 主从复制的格式(mixed,statement,row,默认格式是 statement)
binlog_format=mixed
# 二进制日志自动删除过期的天数。默认值为 0，表示不自动删除
expire_logs_days=7
## 跳过主从复制中遇到的所有错误或指定类型的错误，避免从机复制中断
## 如 1062 错误是指一些主键重复，1032 错误是因为主从数据库数据不一致
slave_skip_errors=1032
# 作为从机时的中继日志
relay_log=demo-mysql-relay-bin
# log_slave_updates=1 表示作为从机时也将复制事件写进自己的二进制日志中
log_slave_updates=1
# 主键自增规则，避免主主同步 ID 重复的问题
# 自增因子（每次加 2）
auto_increment_increment=2
# 自增偏移（从 1 开始），单数
auto_increment_offset=1
```

设置完成后，保存配置，使用如下指令重启数据库服务器：

```
service mysqld restart
```

然后用 root 用户登录服务器，使用如下指令创建一个同步用户并授权：

```
mysql> grant replication slave, replication client on *.* to 'user36'@
'10.10.10.36' identified by 'user123456';
```


其中“user36”为用户名，“user123456”为用户密码。

使用如下指令更新权限，让上面设置立即生效：

```
mysql> flush privileges;
```

使用如下指令查看主机状态：

```
mysql> show master status;
```

查看结果如下所示：

```
+-----+-----+-----+-----+-----+
| File      | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| demo-mysql-bin.000001 | 123 | | mysql | |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

其中，“File”为二进制日志文件名称，“Position”为日志保存位置的偏移量，在后面的从机同步设置中将使用到这两个参数。

2. 从机设置

下面以“10.10.10.36”这台服务器的从机设置为例进行说明。

使用如下指令修改服务器名称：

```
vi /etc/hostname
```

将内容改为“mysql-36”。

使用如下指令修改数据库配置：

```
vi /etc/my.cnf
```

在[mysqld]下面增加以下配置项：

```
[mysqld]
server_id=36
binlog-ignore-db=mysql
log-bin=demo-mysql-bin
```

保存配置，重启数据库服务器。然后，用 root 用户登录数据库，使用如下所示的同步配置：

```
mysql> change master to master_host='10.10.10.35',master_user='user36',
master_password='user123456', master_port=3306, master_log_file='demo-mysql-
bin.000001', master_log_pos=123, master_connect_retry=30;
```

其中，通过“master_log_file”设置了主机的日志文件，通过“master_log_pos”设置

了主机的日志存储位置偏移量。这两个参数必须根据当前主机的状态进行配置。

使用如下指令启动从机，即可开始进行数据同步：

```
mysql>start slave;
```

使用如下指令可以查看从机的同步状态：

```
mysql>show slave status\G;
```

查看结果显示类似于如下所示：

```
***** 1. row *****
Slave_IO_State: Waiting for master to send event
      Master_Host: 10.10.10.35
      Master_User: user36
      Master_Port: 3306
      Connect_Retry: 30
      Master_Log_File: demo-mysql-bin.000001
      Read_Master_Log_Pos: 123
      Relay_Log_File: demo-mysql-relay-bin.000002
      Relay_Log_Pos: 287
      Relay_Master_Log_File: demo-mysql-bin.000001
      Slave_IO_Running: Yes
      Slave_SQL_Running: Yes
      .....
```

在上面结果中，如果“Slave_IO_Running”和“Slave_SQL_Running”都显示为“Yes”，即表示同步成功。

服务器“10.10.10.37”的从机设置可以参照上面方法实现。

13.3 主主同步设置

将两个集群分组的主机互相进行主从同步设置就可以实现主主同步。

首先，参照上节的方法，在集群分组2之中实现主从同步设置。

其中，集群分组2的主机“10.10.10.25”的数据库配置与分组1的数据库配置相似，只是主键的配置为了避免冲突有些不一样，即使用双数作为主键，如下所示：

```
[mysqld]
server_id=25
binlog-ignore-db=mysql
log-bin=demo-mysql-bin
binlog_cache_size=1M
```



```
binlog_format=mixed
expire_logs_days=7
slave_skip_errors=1032
relay_log=demo-mysql-relay-bin
log_slave_updates=1
auto_increment_increment=2
#自增偏移(从2开始), 双数
auto_increment_offset=2
```

集群分组 2 的主从设置完成之后。下面对两个分组的主机实现主主设置, 首先实现分组 1 的主机 (10.10.10.35) 与分组 2 的主机 (10.10.10.25) 的主从设置。

1. 分组 1 的主机配置

在“10.10.10.35”主机上创建同步用户并授权:

```
mysql> grant replication slave, replication client on *.* to 'user25'@
'10.10.10.25' identified by 'user123456';
```

更新权限:

```
mysql> flush privileges;
```

查看主机状态:

```
mysql> show master status;
```

记下查看结果中的日志文件名称和存储位置偏移量。

2. 分组 2 的从机配置

使用如下同步配置:

```
mysql> change master to master_host='10.10.10.35',master_user='user25',
master_password='user123456', master_port=3306, master_log_file='demo-mysql-
bin.000001', master_log_pos=123, master_connect_retry=30;
```

其中, 日志文件名称和存储位置偏移量按上面主机查询的结果填写。

启动从机进行同步:

```
mysql> start slave;
```

查看同步状态:

```
mysql> show slave status\G;
```

在查询结果中如果包含如下所示的两行信息即表示同步设置成功:

```
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
```

现在，反过来以“10.10.10.25”为主机，以“10.10.10.35”为从机，再来设置主从同步，这可参照上面的方法实现。互为主从设置完成之后，就实现了主主同步的设置。

为了对上面的同步设置进行验证，可以在各个主机上创建数据库，再执行一些插入或删除数据的操作，然后在各个从机中查看结果。如果各种操作都能同步成功，即可宣告主主同步和主从同步设置大功告成。

如果出现同步失败的情况，可以先停止失败的从机，视情况更改日志文件名称和偏移量，然后再启动从机继续进行同步。

停止从机可以使用如下的指令：

```
mysql>stop slave;
```

最后需要说明一点的是，在生产实际中，推荐使用 UUID 来作为数据库的主键，这样可以避免主键冲突的情况发生，而且也方便在集群中创建更多的分组。

13.4 数据库代理中间件选择

实现了数据库集群之后，已经解决了数据库的单机服务器的性能瓶颈问题，并且也建立了高可用的分布式架构，那么对于应用程序和数据库客户端，应该怎么调用数据库才能更好地使用这种高可用和高性能的分布式集群系统呢？这就要借助数据库代理中间件来实现了。

MySQL 的数据库代理服务中间件有很多，而且大多数是开源的，例如，MyCat、Proxy、Amoeba、OneProxy 等，其中比较优秀的是 MyCat 和 OneProxy。

MyCat 在大流量访问中有极佳的性能表现，它是用 Java 语言开发的，配置文件使用 XML 的形式，稍显复杂，特别是它的分区表的配置显得有点累赘，并且一些用户对它的稳定性也颇有微词，所以这里推荐使用 OneProxy。

OneProxy 是基于 MySQL 官方的 Proxy 中间件的设计思想开发的，运行稳定性好，配置也较为简单，分区表与 MySQL 分区表设置根本上是一致的。虽然是一个收费的商业软件，但也提供了免费的社区版可以使用。

13.5 使用 OneProxy 实现读写分离设计

OneProxy 可以非常方便地使用 MySQL 的集群体系架构,既可以按数据库的集群分组来实现高可用设计,也可以按主从同步轻易实现读写分离设计。使用两个集群分组的 OneProxy 调用设置的网络结构如图 13-4 所示。

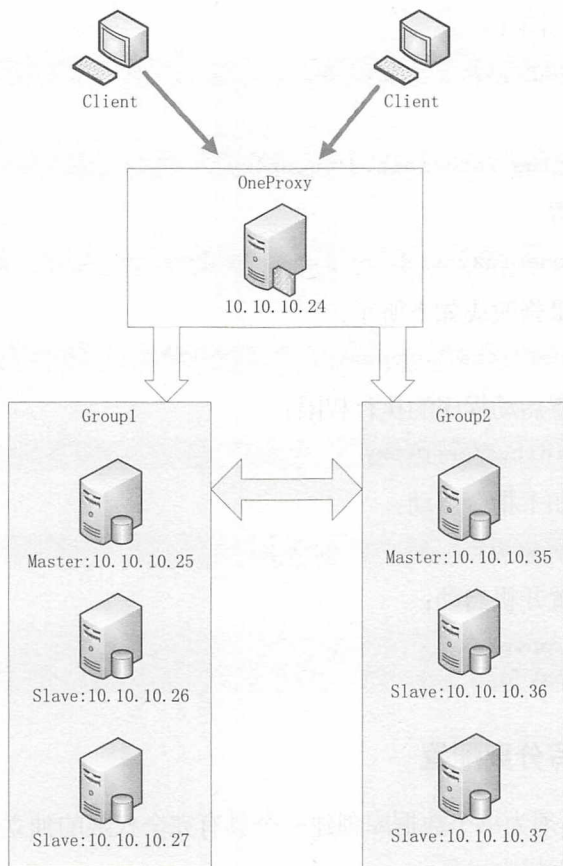


图 13-2 OneProxy 调用设置网络结构图

13.5.1 OneProxy 安装

可以通过官方网站下载 OneProxy 的安装包:

<http://www.onexsoft.com/>

下列安装以“6.0.0”的版本为例进行说明。我们将 OneProxy 安装在 IP 为“10.10.10.24”

的机器上。

下载安装包后解压缩：

```
tar xf oneproxy-rhel5-linux64-v6.0.0-ga.tar.gz
```

将程序文件移动到“/usr/local/”目录之中：

```
mv oneproxy /usr/local/oneproxy
```

切换到 **oneproxy** 目录：

```
cd /usr/local/oneproxy
```

创建启动程序：

```
cp oneproxy.service /etc/init.d/oneproxy
```

修改启动程序配置：

```
vi /etc/init.d/oneproxy
```

将其中的工作目录修改成如下所示：

```
ONEPROXY_HOME=/usr/local/oneproxy
```

保存修改后，设置启动程序的执行权限：

```
chmod a+x /etc/init.d/oneproxy
```

现在，可以使用如下指令启动：

```
service oneproxy start
```

使用如下指令设置开机启动：

```
chkconfig --add oneproxy  
chkconfig oneproxy on
```

13.5.2 高可用读写分离配置

使用 **OneProxy** 必须为每个数据库创建一个具有完全权限的独立用户。在创建用户时，必须在数据库集群的主机中进行。

例如，对于订单服务的数据库“**orderdb**”，可以使用如下指令来创建用户并授权：

```
mysql> grant all privileges on orderdb.* to 'orderuser'@'%' identified by  
'12345678' with grant option;
```

即创建一个具有完全权限的用户，其用户名为“**orderuser**”，密码为“**12345678**”，并设置在任何地方都可以访问，这里指在我们的安全的局域网之中。

需要注意的是，上面授权的用户将保存在数据库“**mysql**”的“**user**”表之中，虽然我

们已经在主从设置中忽略了数据库“mysql”的同步,但是在创建用户时,并没有使用“use”指令切换过数据库,所以上面创建的用户还会在各个从机上进行同步,这倒为我们省略了很多操作。如果要禁止这种同步,可以在主从设置中对主机的数据库配置增加一个忽略对“user”表进行同步的配置,那么,如果那样的话,给数据库授权的指令就必须在每个服务器上都执行一遍。

在 OneProxy 的服务器中,假设还在目录“/usr/local/oneproxy”之中,使用如下指令为用户密码生成加密串:

```
./bin/mysqlpwd 12345678
```

执行后将生成如下所示已经加密的密码字符串:

```
40739ED24B5DC118DC16397AB14E64C680637C0D
```

使用如下指令编辑 OneProxy 配置:

```
vi ./conf/proxy.conf
```

如下所示为使用两个集群分组的读写分离配置:

```
[oneproxy]
keepalive = 1
event-threads = 4
log-file = log/oneproxy.log
pid-file = log/oneproxy.pid
lck-file = log/oneproxy.lck
mysql-version = 5.7.19
proxy-address = :3306
proxy-master-addresses.1 = 10.10.10.35:3306@group1
proxy-master-addresses.2 = 10.10.10.25:3306@group2
proxy-slave-addresses.1 = 10.10.10.36:3306@group1
proxy-slave-addresses.2 = 10.10.10.37:3306@group1
proxy-slave-addresses.3 = 10.10.10.26:3306@group2
proxy-slave-addresses.4 = 10.10.10.27:3306@group2
proxy-user-list.1 = orderuser/40739ED24B5DC118DC16397AB14E64C680637C0D
@orderdb
proxy-user-list.2 = merchantuser/40739ED24B5DC118DC16397AB14E64C680637C0D
@merchantdb
proxy-part-tables.1 = /usr/local/oneproxy/conf/part1.txt
proxy-charset = utf8_bin
proxy-group-policy.1 = group1:read_balance
proxy-group-policy.2 = group2:read_balance
proxy-group-security.1 = group1:0
proxy-group-security.2 = group2:0
```

```

proxy-security-level = 0
proxy-sequence.1 = default
#监控端口
proxy-httpserver = :8080
#自动剔除节点
proxy-replication-check = 1
proxy-httptitle = OneProxy Monitor

```

其中，只配置了订单服务的数据库“orderdb”的访问用户“orderuser”和商家服务的数据库“merchantdb”的访问用户“merchantuser”，其他服务的数据库可以参照上面方法增加进来。

保存配置后，记得必须使用如下指令设定配置文件的读取权限：

```
chmod 660 conf/proxy.conf
```

上面各项配置的参数含义可以参照如表 13-3 所示的说明。

表 13-3 配置参数列表

参 数 名	含 义
proxy-address	Proxy Server 自身监听地址
proxy-master-addresses	Master 节点地址（可写入节点）
proxy-slave-addresses	Slave 节点地址（可读取节点）
proxy-user-list	Proxy 用户列表（用户名：口令）
proxy-table-map	为某张表指定“Server Group”
proxy-sql-review	为某张表指定 Where 条件中必需的列
proxy-database	Proxy 对应的后端数据库，默认为 test
proxy-charset	Proxy 字符集，默认为 utf8_general_ci
proxy-lua-script	Proxy 功能脚本（非常重要）
proxy-group-policy	预定义策略：0 代表由 Lua Script 来决定，默认为 Master Only；1 代表 Read Failover；2 代表 Read/Write Split（Master 节点不参与读操作）；3 代表双 Master 结构或者 XtraDB Cluster 结构，即多主对等的方式；4 代表 Read/Write Split（Master 节点共同参与读操作）；5 代表读写随机
proxy-security-level	安全级别：0 为默认值；1 表示禁止 DDL；2 表示禁止不带条件的查询语句；3 表示只允许 SELECT
proxy-group-security	为特定 Server Group 设置安全级别 安全级别：0 为默认值，1 表示禁止 DDL，2 表示禁止不带条件的查询语句，3 表示只允许 SELECT
event-threads	并发线程数，最大允许 48 个线程

OneProxy 还有一个管理后台，OneProxy 启动之后，可以通过 MySQL 客户端进行登

录。管理后台的默认端口是 4041，用户名为 admin，密码为 OneProxy。

例如，可以在安装了 MySQL 的机器上使用如下指令登录：

```
mysql -u admin -h 10.10.10.24 -P4041 -pOneProxy
```

登录管理后台之后，可以执行如表 13-4 所示的一些指令。

表 13-4 管理接口指令列表

命 令	描 述	例 子
LIST HELP	列出所有指令	list help
LIST BACKEND	列出所有后端数据库	list backend
LIST GROUP	列出所有的 server group，具体含义请参考重要概念	list group
LIST POOL	列出 OneProxy 与每个后端数据库建立的连接池大小与连接池配置	list pool
LIST QUEUE	列出每个队列里到达的请求数量与已处理完成的请求	list queue
LIST THREADS	列出每个线程处理过的请求数	list threads
LIST TABLEMAP	列出 table 与 server group 的对应关系	list tablemap
LIST USERS	列出用户	list users
LIST SQLSTATS	列出执行过的 SQL 统计	list sqlstats [hash]
LIST SQLTEXT	列出执行过的 SQL 与 hashcode 的对应关系	list sqltext [hash]
SET MASTER	指定某后端 DB 为写库	set master '192.168.1.119:3306'
SET SLAVE	指定某后端 DB 为读库	set slave '192.168.1.119:3306'
SET OFFLINE	下线指定的后端数据库	set offline '192.168.1.119:3306'
SET ONLINE	上线指定的后端数据库	set online '192.168.1.119:3306'
SET GPOLICY	指定 server group 的策略 预定义策略： 0 代表由 Lua Script 来决定，默认为 Master Only 1 代表 Read Failover 2 代表 Read/Write Split（Master 节点不参与读操作） 3 代表双 Master 结构或者 XtraDB Cluster 结构，即多主对等的方式 4 代表 Read/WriteSplit（Master 节点共同参与读操作） 5 代表读写随机	set gpolicy default 1
SET GMASTER	针对 XtraDB Cluster，指定某个编号的数据库为写库	set gmaster default 1
SET GACCESS	指定 server group 允许的 SQL 类型： 0（缺省值）代表无任何限制 1 代表禁止 DDL 操作 2 代表禁止不带 Where 条件的 Select、Update 或 Delete	set gaccess default 1

续表

命 令	描 述	例 子
SET POOLMIN	设置 OneProxy 与后端数据库连接池的最少连接数	set poolmin '192.168.1.119:3306' 5
SET POOLMAX	设置 OneProxy 与后端数据库连接池的最大连接数, 实际的连接数可以超过这个数值,	set poolmax '192.168.1.119:3306' 300
SET SQLSTATS	打开、关闭或者清空 SQL 统计	set sqlstats {on off clear}
MAP	把某个表归属给某个 server group	map my_test1_0 default
UNMAP	删除表与 server group 的映射关系	unmap my_test1_0
SUSPEND	让 event process 停止指定秒数	suspend 10
SHUTDOWN	关闭 proxy	shutdown force

注意: 在使用粗体部分的指令前, 需要了解指令对系统的含义, 否则可能导致数据不一致或者系统不可用。

另外, 还可以通过监控端口来查看各个数据库服务器的连接情况, 使用如下所示的链接可以打开监控的控制台:

```
http://10.10.10.24:8080
```

其他有关 OneProxy 配置的详细说明还可以参考友哥 (OneProxy 的开发者) 的博客:

```
https://www.cnblogs.com/youge-OneSQL/articles/4208583.html
```

13.6 OneProxy 分库分区设计

对于超大容量的表存储来说, MySQL 支持分区表设计, 可以按某一字段进行按范围 (Range)、按值 (List)、按哈希算法 (Hash) 等方法进行分区。

而 OneProxy 将分区表的概念从数据库层抽象到了 SQL 的转发器层中, 然后对通信协议进行分析, 可以根据 SQL 查询语句的表名及传入参数进行对上层应用透明的智能路由, 从而实现了虚拟分区表的效果, 这种分区的功能对应用来说是完全透明的。

在 OneProxy 中同样也支持按范围 (Range)、按值 (List) 和按哈希算法 (Hash) 进行虚拟分库分表的设计, 从内容上看与 MySQL 创建分区表的关键信息非常类似。

下面分别对这三种分区方法的分库分表配置进行说明。

13.6.1 按范围分库分表

按范围 (Range) 分库分表的配置, 必须有一个针对应用的虚拟表名 (Table), 并指定一个用于分区的字段 (PKey)、字段的类型 (Type) 及分区的方法 (Method)。同时针对每一个分区, 可以使用后缀 (Suffix) 来设置独立的表名, 并且指定分区所在的集群分组 (Group), 以及分区字段取值的上限 (Value) 等配置。

例如, 如下所示为一个订单表的分区表设计:

```
[
  {
    "table"   : "t_order",
    "pkey"    : "id",
    "type"    : "int",
    "method"  : "range",
    "partitions":
      [
        { "suffix" : "_0", "group": "group1", "value" : "100000" },
        { "suffix" : "_1", "group": "group1", "value" : "200000" },
        { "suffix" : "_2", "group": "group2", "value" : "300000" },
        { "suffix" : "_3", "group": "group2", "value" : null      }
      ]
  }
],
```

这样, 当访问虚拟表名 “t_order” 的时候, 按其 ID “value” 的范围将其导向到真实表名如 “t_order_0”、“t_order_1” 等分表之中进行数据的存取操作。

13.6.2 按值分库分表

按值 (List) 进行分库分表的配置, 也是在虚拟表名中指定一个用于分区的字段 (PKey)、字段的类型 (Type) 及分区的方法 (Method)。同时针对每一个分区, 使用后缀 “Suffix” 来设置独立的表名, 并且指定分区所在的集群分组 (Group), 以及分区字段能取的值列表 (Value) 等配置, 当一个分区没有指定任何分区值列表时, 表示所有其他的值都落入这个分区之中。

例如, 如下所示为订单按值进行分区的设计:

```
[
  {
    "table"   : "t_order",
    "pkey"    : "id",
```

```

    "type"    : "int",
    "method"  : "list",
    "partitions":
    [
        { "suffix" : "_0", "group": "group1", "value" : ["1","2","3"] },
        { "suffix" : "_1", "group": "group1", "value" : ["4","5","6"] },
        { "suffix" : "_2", "group": "group2", "value" : ["7","8","9"] },
        { "suffix" : "_3", "group": "group2", "value" : ["10","11","12"] },
        { "suffix" : "_4", "group": "group2", "value" : [] }
    ]
}
]

```

其中，真实表名由虚拟表名和后缀组成，例如，“t_order_0”，并且数据的存取将分别分到指定的集群分组中进行。

13.6.3 按哈希算法分库分表

按哈希算法(Hash)分库分表的配置数据，也必须有一个针对应用的虚拟表名(Table)，并指定一个用于分区的字段(PKey)、字段的类型(Type)及分区的方法(Method)。同时针对每一个分区可使用增加后缀(Suffix)的方法设置独立的表名，并且指定分区所在的集群分组(Group)。需要注意的是，哈希分区并不需要为每个分区指定值范围或值列表，而是由OneProxy里的哈希算法根据分区数进行自动计算的，哈希分区里的分区数量不能随便调整。

例如，如下所示为订单的哈希分区设计：

```

[
  {
    "table" : "t_order",
    "pkey"  : "id",
    "type"  : "int",
    "method" : "hash",
    "partitions" :
    [
      { "suffix" : "_0", "group": "group1" },
      { "suffix" : "_1", "group": "group2" },
      { "suffix" : "_2", "group": "group1" },
      { "suffix" : "_3", "group": "group2" }
    ]
  }
]

```


其中，真实表名由虚拟表名及其后缀所组成，并且分别存储在指定的不同集群分组之中。例如，“t_order_0”存储于分组“group1”之中，“t_order_1”存储于分组“group2”之中。

分区的配置都是使用文本文件来实现的，如果要在一个文件中使用多个分区配置，可以使用如下所示的格式进行配置：

```
[
  {
    "table" : "t_goods",
    "pkey" : "id",
    "type" : "int",
    "method" : "hash",
    "partitions" :
    [
      { "suffix" : "_0", "group": "group1" },
      { "suffix" : "_1", "group": "group2" },
      { "suffix" : "_2", "group": "group1" },
      { "suffix" : "_3", "group": "group2" }
    ]
  },
  {
    "table" : "t_order",
    "pkey" : "id",
    "type" : "char",
    "method" : "hash",
    "partitions" :
    [
      { "suffix" : "_0", "group": "group1" },
      { "suffix" : "_1", "group": "group2" },
      { "suffix" : "_2", "group": "group1" },
      { "suffix" : "_3", "group": "group2" }
    ]
  }
]
```

文件设置完成之后，使用如下所示的方式加进 OneProxy 的配置之中：

```
proxy-part-tables.1 = /usr/local/oneproxy/conf/part1.txt
```

在上面几种分表方法中，建议使用哈希算法，这样可以提供更好的性能。

13.7 双机热备设计

使用了数据库的代理服务中间件之后，我们实现了高性能的读写分离配置，但是，对于代理服务器本身还存在着一个单点故障的问题。

OneProxy 已经具有高可用的设计，通过配置可以使用 VIP 的方式来构建一个双机热备设计，如图 13-3 所示。

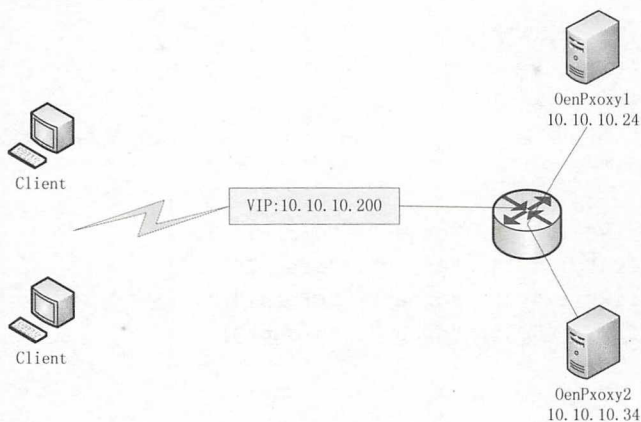


图 13-3 OneProxy 双机热备网络拓扑

如果我们使用的两台 OneProxy 代理服务器的 IP 和 VIP 如下所示：

```
OneProxy1: 10.10.10.24
OneProxy2: 10.10.10.34
VIP: 10.10.10.200
```

即可分别在两台服务器中使用如下方法在原来的 proxy.conf 配置基础上增加双机热备的配置。

OneProxy1 的 proxy.conf 增加如下所示的配置：

```
remote-address = 10.10.10.34:4041
vip-address = 10.10.10.200/ ens192:0
```

OneProxy2 的 proxy.conf 增加如下所示的配置：

```
remote-address = 10.10.10.24:4041
vip-address = 10.10.10.200/ ens192:0
```

完成上面的配置后，即可启动两台服务器的 OneProxy 服务，提供高可用服务。假如其中的 VIP 未启动，则可以使用如下方法进行启动。

启动 VIP:

```
ifconfig ens192:0 10.10.10.100 broadcast 10.10.10.100 netmask 255.255.255.255 up
```

然后, 使用如下指令查看 VIP 的启动结果:

```
ip addr
```

如果在输出结果中能看到 VIP 的设置地址 “10.10.10.100”, 则表示启动成功, 如下所示:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:81:6a:85 brd ff:ff:ff:ff:ff:ff
    inet 10.10.10.26/24 brd 10.10.10.255 scope global ens192
        valid_lft forever preferred_lft forever
    inet 10.10.10.100/32 brd 10.10.10.100 scope global ens192:0
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe81:6a85/64 scope link
        valid_lft forever preferred_lft forever
```

如果要关闭 VIP, 则可以使用如下指令:

```
ifconfig ens192:0 down
```

启动了双机热备的服务之后, 在应用中可以将数据源的配置中连接服务器的地址改为 VIP 的地址。

13.8 小结

本章介绍了在 CentOS 7 中安装 MySQL 的简易方法, 并使用主从设计构建了分布式的数据库管理系统, 搭建了一个高性能并且可扩展的数据库集群体系, 同时使用分组的方式实现了高可用的设计。在数据库访问的设计中, 使用 OneProxy 中间件实现了可配置的读写分离调用方法, 并结合分库分区表的功能来提高数据库的访问效率。最后, 使用双机热备设计, 为数据库的使用提供了更加安全可靠的保障。

下一章将介绍分布式文件系统及其他一些基础服务的安装和配置的方法。

分布式文件系统等基础设施 安装与配置

传统的 Web 应用的文件管理方式，例如，图片和视频文件的上传和使用等，大多数是将文件存储在服务器本地，这种方式已经不能适应微服务应用的使用。一方面，微服务应用发布在分布式环境中，随时随地都可以进行多副本的部署，所以它的媒体文件的存放必须有一个统一的地方；另一方面，建立一个独立而高效的文件系统，也是高可用和高性能的应用平台一个有机组成部分。

我们将使用开源的 FastDFS 这个轻量级的分布式文件系统，搭建一个高可用并且可以持续扩展的分布式文件系统。

本章除介绍 FastDFS 的安装和配置外，同时也对应用平台的其他几个基础设施的安装进行了简要说明，分别是 GitLab、Redis 和 RabbitMQ。

14.1 高可用的分布式文件系统构建

FastDFS 由跟踪器（Tracker）和存储节点（Storage）两部分组成。跟踪器用来调度来自客户端的请求，并记录存储服务器的信息。存储节点保存文件及其属性，同时进行文件同步处理工作。文件的存储使用分组（或分卷）的方式进行组织。搭建两个以上的跟踪器可以组成一个高可用的分布式文件系统，它的架构设计图如图 14-1 所示。

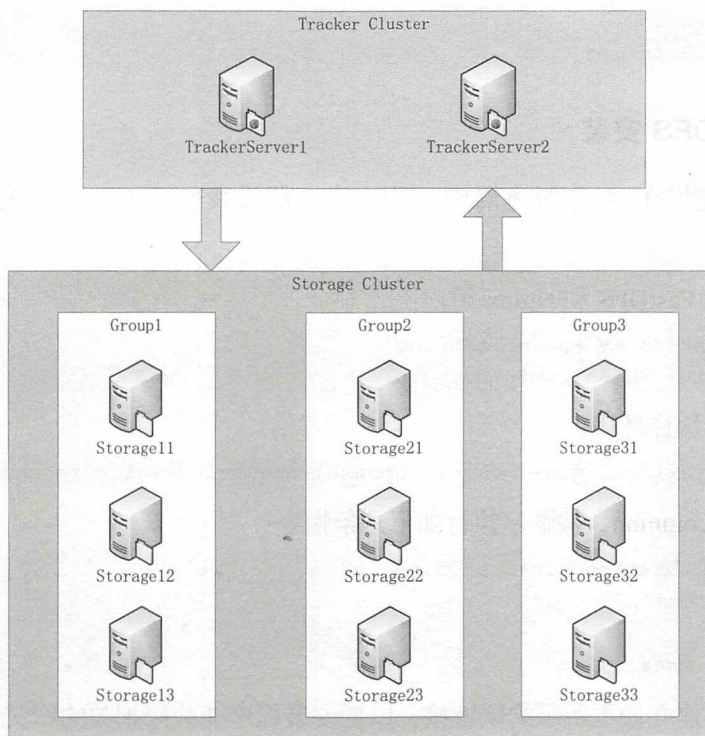


图 14-1 高可用分布式文件系统架构设计图

基于如图 14-1 所示的架构设计，我们将使用 4 台服务器来搭建一个高可用的分布式文件系统，如下所示：

```

Tracker Server1: 10.10.10.22
Tracker Server2: 10.10.10.32
Storage1: 10.10.10.23
Storage2: 10.10.10.33

```

在这种架构设计中，增加存储节点即可提高和扩展文件的存取性能。安装了分布式文件系统之后，我们还将使用 Nginx 为 Web 端的文件访问搭建一个负载均衡服务。

安装过程中使用的安装包都可以从如下所示的网址中下载：

```

https://github.com/happyfish100/libfastcommon/releases/tag/V1.0.35
https://github.com/happyfish100/fastdfs/releases
http://nginx.org/en/download.html

```

我们将用到如下所示的一些安装包：

```

libfastcommon-1.0.35.tar.gz
fastdfs-5.10.tar.gz

```

```
pcre-8.40.tar.gz
nginx-1.10.3.tar.gz
```

14.1.1 FastDFS 安装

下列安装过程在 4 个服务器中都要完成。假设我们把下载的安装包都放在目录“/opt”中。

首先，创建 FastDFS 和 Nginx 用户：

```
useradd fastdfs -M -s /sbin/nologin
useradd nginx -M -s /sbin/nologin
```

然后，安装编译环境：

```
yum -y install gcc gcc+ gcc-c++ openssl openssl-devel pcre pcre-devel
```

安装 libfastcommon，按顺序执行如下几条指令：

```
tar -zxvf libfastcommon-1.0.35.tar.gz
cd libfastcommon-1.0.35
./make.sh
./make.sh install
```

完成之后，建立如下所示的软链接，以便安装程序能找到相关的库文件：

```
ln -sv /usr/include/fastcommon /usr/local/include/fastcommon
ln -sv /usr/include/fastdfs /usr/local/include/fastdfs
ln -sv /usr/lib64/libfastcommon.so /usr/local/lib/libfastcommon.so
```

按顺序执行如下命令，安装 FastDFS：

```
tar -zxvf fastdfs-5.10.tar.gz
cd fastdfs-5.10
./make.sh
./make.sh install
```

安装结束将输出成功标志，接下来配置各个不同的服务。

14.1.2 跟踪服务器配置

在 Tracker Server 机器上创建数据存储目录：

```
mkdir -p /data/fastdfs/tracker
```

顺序执行如下指令，编辑“tracker.conf”配置：

```
cd /etc/fdfs
cp tracker.conf.sample tracker.conf
```



```
vi /etc/fdfs/tracker.conf
```

在配置文件中编辑如下各项配置：

```
#绑定 IP，为空表示使用本机 IP
bind_addr=
#端口
port=22122
#连接超时时间
connect_timeout=30
#日志数据路径
base_path=/data/fastdfs/tracker
#上传文件时选择 group 的方法
#0:轮询，1:指定组，2:选择剩余空间最大
store_lookup=2
#如果上面的配置是 1，那么这里必须指定组名
store_group=group2
#上传文件时选择 server 的方法
#0:轮询，1:按 IP 地址排序，2:通过权重排序
store_server=0
#storage 上预留空间
reserved_storage_space = 10%
http.server_port=8080
```

保存并退出，然后使用如下指令启动 Tracker Server：

```
service fdfs_trackerd start
```

使用如下指令可以查看 Tracker Server 监听的端口：

```
netstat -unltp|grep fdfs
```

14.1.3 存储节点配置

在 Storage 服务器上，创建如下所示的数据存储目录：

```
mkdir -p /data/fdfs_storage/base
mkdir -p /data/fdfs_storage/storage0
mkdir -p /data/fdfs_storage/storage1
```

按顺序执行如下指令，编辑存储节点配置：

```
cd /etc/fdfs
cp storage.conf.sample storage.conf
vi storage.conf
```

按如下所示编辑各项内容：

```
#storage server 所属组名
```



```

group_name=group1
#绑定 IP
bind_addr=
#storage server 的端口
port=23000
#连接超时时间
connect_timeout=30
#日志数据路径
base_path=/data/fdfs_storage/base
#storage path 的个数
store_path_count=2
#根据 store_path_count 的值, 就要有 storage0 到 storage (N-1) 个路径
store_path0=/data/fdfs_storage/storage0
store_path1=/data/fdfs_storage/storage1
#跟踪服务器
tracker_server=10.10.10.22:22122
tracker_server=10.10.10.32:22122

```

保存编辑后, 使用如下指令启动存储节点:

```
service fdfs_storaged start
```

使用如下指令可检查服务运行状态:

```
/usr/bin/fdfs_monitor /etc/fdfs/storage.conf
```

14.1.4 上传文件测试

现在回到 Tracker Server 机器上 (例如 10.10.10.22), 可以使用如下方法进行文件上传测试。

顺序执行如下指令, 编辑客户端配置:

```

cd /etc/fdfs
cp client.conf.sample client.conf
vi /etc/fdfs/client.conf

```

修改如下各项内容:

```

base_path=/data/fastdfs/tracker #tracker 服务器文件路径
tracker_server=10.10.10.22:22122 #tracker 服务器 IP 地址和端口号
# tracker 服务器的 http 端口号, 必须和 tracker 的设置一样
http.tracker_server_port=8080

```

假如在 “/opt” 中存在一个图片文件为 “1.png”, 即可以使用如下指令上传文件:

```
/usr/bin/fdfs_upload_file /etc/fdfs/client.conf /opt/1.png
```



上传成功将返回文件信息，类似于如下所示：

```
group1/M00/00/00/wKgBGFk3kUqACu9JAAGmMprynZs674.png
```

14.1.5 Nginx 安装及负载均衡配置

分布式文件系统安装完成之后，已经可以在应用程序中进行调用，但是要想在 Web 端的页面上进行访问，还必须借助 Nginx 来提供访问服务。使用 Nginx 不但可以构建负载均衡服务，还能使用缓存设置。跟踪器和存储节点都必须安装 Nginx，下面分别进行说明。

因为 Web 是在外网中进行访问的，所以必须在防火墙的流量策略中新建如下几条规则，这样才能在外网中访问相应局域网中的服务：

```
192.168.1.22:80→10.10.10.22:80
192.168.1.32:80→10.10.10.32:80
192.168.1.23:80→10.10.10.23:80
192.168.1.33:80→10.10.10.33:80
```

1. 在跟踪器上安装 Nginx

在两台 Tracker Server 机器上，都按如下方法来安装和配置 Nginx。

安装 pcre 支持库，按顺序执行如下指令：

```
tar xf pcre-8.40.tar.gz
cd pcre-8.40
./configure --prefix=/usr/local/pcre
make && make install
```

完成后返回安装包存放路径“/opt”，按顺序执行如下指令，安装 Nginx：

```
tar xf nginx-1.10.3.tar.gz

cd nginx-1.10.3

./configure --prefix=/data/nginx \
--with-pcre=/opt/pcre-8.40 \
--user=nginx \
--group=nginx \
--with-http_ssl_module \
--with-http_realip_module \
--with-http_stub_status_module

make && make install
```

编辑 Nginx 的配置文件“nginx.conf”，内容如下所示：



```

user nginx nginx;
worker_processes 2;
#pid /usr/local/nginx/nginx.pid;
worker_rlimit_nofile 51200;
events
{
    use epoll;
    worker_connections 20480;
}

http
{
    include mime.types;
    default_type application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] "$request"'
        '$status $body_bytes_sent "$http_referer"'
        '"$http_user_agent" "$http_x_forwarded_for" "$request_time"';

    access_log /data/nginx/logs/access.log main;

    upstream server_group1{
        server 192.168.1.23;
        server 192.168.1.33;
    }

    server {
        listen 80;
        server_name localhost;
        location /group1 {
            #include proxy.conf;
            proxy_pass http://server_group1;
        }
    }
}

```

这个配置的原理是对两个存储节点上 HTTP 服务的访问，将由跟踪服务器进行负载均衡调度。例如，上节测试生成的图片文件可以通过跟踪服务器使用如下所示的链接进行访问：

```
http://192.168.1.22/group1/M00/00/00/wKgBGfk3kUqACu9JAAGmMprynZs674.png
```

如果使用了域名指向，则可以将上面配置的“localhost”改为相应的域名。

使用如下指令启动 Nginx：

```
/data/nginx/sbin/nginx
```



2. 存储节点上安装 Nginx

在两台 Storage 机器上, 进入存放安装包的路径 “/opt”, 然后按顺序执行如下指令, 安装 pcre 支持库:

```
tar xf fastdfs-nginx-module_v1.16.tar.gz

tar xf pcre-8.40.tar.gz
cd pcre-8.40

./configure --prefix=/data/pcre

make && make install
```

完成后, 返回存放安装包的路径 “/opt”, 按顺序执行如下指令安装 Nginx:

```
tar xf nginx-1.10.3.tar.gz

cd nginx-1.10.3

./configure --prefix=/data/nginx \
--with-pcre=/opt/pcre-8.40 \
--user=nginx \
--group=nginx \
--with-http_ssl_module \
--with-http_realip_module \
--with-http_stub_status_module \
--add-module=/opt/fastdfs-nginx-module/src

make && make install
```

完成后按如下指令复制相关文件:

```
cp /opt/fastdfs-nginx-module/src/mod_fastdfs.conf /etc/fdfs/
cd /opt/fastdfs-5.10/conf
cp anti-steal.jpg http.conf mime.types /etc/fdfs/
```

完成后使用如下指令编辑配置文件 “mod_fastdfs.conf”:

```
vi /etc/fdfs/mod_fastdfs.conf
```

编辑如下各项内容:

```
#日志目录
base_path=/tmp
#跟踪服务器
tracker_server=10.10.10.22:22122
tracker_server=10.10.10.32:22122
```



```
#url 中是否有 group 名称
url_have_group_name = true
#storage path 的个数
store_path_count=2
#根据 store_path_count 的值, 就要有 storage0 到 storage(N-1) 个路径
store_path0=/data/fdfs_storage/storage0
store_path1=/data/fdfs_storage/storage1
```

使用如下指令编辑 Nginx 配置文件:

```
vi /data/nginx/conf/nginx.conf
```

文件的内容如下所示:

```
user nginx nginx;
worker_processes 2;
#pid /usr/local/nginx/logs/nginx.pid;
worker_rlimit_nofile 1024;

events {
    use epoll;
    worker_connections 1024;
}

http {

    include mime.types;
    server_names_hash_bucket_size 128;
    client_header_buffer_size 32k;
    large_client_header_buffers 4 32k;
    client_max_body_size 20m;
    limit_rate 1024k;

    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /data/nginx/logs/access.log main;

    server {

        listen 80;
        server_name localhost;

        location ~ /group[0-9]/M00{
```




```

        #root /data/fdfs_storage;
        ngx_fastdfs_module;
    }
}
}

```

保存文件后，使用如下指令创建两个软链接：

```

ln -s /data/fdfs_storage/storage0 /data/fdfs_storage/storage0/M00
ln -s /data/fdfs_storage/storage1 /data/fdfs_storage/storage1/M00

```

然后，使用如下指令启动 Nginx：

```
/data/nginx/sbin/nginx
```

14.1.6 开机启动设置

为了方便运维管理，上面安装的各个服务都可以进行开机启动设置。

1. Tracker 开机启动设置

在 Tracker Server 服务器中创建服务启动文件：

```
vi /etc/rc.d/init.d/fdfs_trackerd
```

输入如下内容：

```

#!/bin/bash
#
# fdfs_trackerd Starts fdfs_trackerd
#
#
# chkconfig: 2345 99 01
# description: FastDFS tracker server
### BEGIN INIT INFO
# Provides: $fdfs_trackerd
### END INIT INFO

# Source function library.
. /etc/init.d/functions

PRG=/usr/bin/fdfs_trackerd
CONF=/etc/fdfs/tracker.conf

if [ ! -f $PRG ]; then
    echo "file $PRG does not exist!"
    exit 2

```



```
fi

if [ ! -f $CONF ]; then
    echo "file $CONF does not exist!"
    exit 2
fi

CMD="$PRG $CONF"
RETVAL=0

start() {
    echo -n "Starting FastDFS tracker server: "
    $CMD &
    RETVAL=$?
    echo
    return $RETVAL
}

stop() {
    $CMD stop
    RETVAL=$?
    return $RETVAL
}

rhstatus() {
    status fdfs_trackerd
}

restart() {
    $CMD restart &
}

case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
        rhstatus
        ;;
    restart|reload)
        restart
        ;;
    condrestart)
        restart
    esac
```




```
;;
*)
    echo $"Usage: $0 {start|stop|status|restart|condrestart}"
    exit 1
esac

exit $?
```

然后，按顺序执行如下指令，设置为开机启动：

```
chmod 755 /etc/rc.d/init.d/fdfs_trackerd
chkconfig --add fdfs_trackerd
chkconfig fdfs_trackerd on
```

2. Storage 开机启动设置

在 Storage 服务器中，创建服务启动文件：

```
vi /etc/init.d/fdfs_storaged
```

输入如下内容：

```
#!/bin/bash
#
# fdfs_storaged Starts fdfs_storaged
#
#
# chkconfig: 2345 99 01
# description: FastDFS storage server
### BEGIN INIT INFO
# Provides: $fdfs_storaged
### END INIT INFO

# Source function library.
. /etc/init.d/functions

PRG=/usr/bin/fdfs_storaged
CONF=/etc/fdfs/storage.conf

if [ ! -f $PRG ]; then
    echo "file $PRG does not exist!"
    exit 2
fi

if [ ! -f $CONF ]; then
    echo "file $CONF does not exist!"
    exit 2
```

```
fi

CMD="$PRG $CONF"
RETVAL=0

start() {
    echo -n "Starting FastDFS storage server: "
    $CMD &
    RETVAL=$?
    echo
    return $RETVAL
}

stop() {
    $CMD stop
    RETVAL=$?
    return $RETVAL
}

rhstatus() {
    status fdfs_storaged
}

restart() {
    $CMD restart &
}

case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
        rhstatus
        ;;
    restart|reload)
        restart
        ;;
    condrestart)
        restart
        ;;
    *)
        echo "Usage: $0 {start|stop|status|restart|condrestart}"
        exit 1
esac
```



```
exit $?
```

按顺序执行如下指令，设为开机启动：

```
chmod 755 /etc/rc.d/init.d/fdfs_storaged
chkconfig --add fdfs_storaged
chkconfig fdfs_storaged on
```

3. Nginx 开机启动设置

在 4 台服务器中都创建一个 Nginx 的启动文件：

```
vi /etc/init.d/nginx
```

输入如下内容：

```
#!/bin/bash
# chkconfig: - 85 15
PATH=/data/nginx
DESC="nginx daemon"
NAME=nginx
DAEMON=$PATH/sbin/$NAME
CONFIGFILE=$PATH/conf/$NAME.conf
PIDFILE=$PATH/logs/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
set -e
[ -x "$DAEMON" ] || exit 0
do_start() {
$DAEMON -c $CONFIGFILE || echo -n "nginx already running"
}
do_stop() {
$DAEMON -s stop || echo -n "nginx not running"
}
do_reload() {
$DAEMON -s reload || echo -n "nginx can't reload"
}
case "$1" in
start)
echo -n "Starting $DESC: $NAME"
do_start
echo "."
;;
stop)
echo -n "Stopping $DESC: $NAME"
do_stop
echo "."

```

```
;;
reload|graceful)
echo -n "Reloading $DESC configuration..."
do_reload
echo "."
;;
restart)
echo -n "Restarting $DESC: $NAME"
do_stop
do_start
echo "."
;;
*)
echo "Usage: $SCRIPTNAME {start|stop|reload|restart}" >&2
exit 3
;;
esac
exit 0
```

按顺序执行如下指令，设为开机启动：

```
chmod 755 /etc/rc.d/init.d/nginx
chkconfig --add nginx
chkconfig nginx on
```

上面一些脚本文件可以从如下网址取得：

<https://gitee.com/chenshaojian/SpringCloud>

14.2 GitLab 安装

微服务的配置管理中心使用了 Git 代码仓库来存放配置文件，所以这里介绍一下 GitLab 的安装。GitLab 的社区版是免费的，并且安装也很简单。

使用如下指令安装 SSH，并启动 SSH 服务：

```
sudo yum install curl policycoreutils openssh-server openssh-clients
sudo systemctl enable sshd
sudo systemctl start sshd
```

安装一个邮件系统用来发送邮件：

```
sudo yum install postfix
sudo systemctl enable postfix
sudo systemctl start postfix
```


安装 GitLab 社区版:

```
curl -sS http://packages.gitlab.cc/install/gitlab-ce/script.rpm.sh | sudo
bash
sudo yum install gitlab-ce
```

编辑配置:

```
vi /etc/gitlab/gitlab.rb
```

找到如下配置项并按提示更改:

```
#GitLab 的首页地址
external_url 'http://10.10.10.20'

### Email Settings

gitlab_rails['gitlab_email_from'] = 'username@youremail.domain'
user['git_user_email'] = 'username@youremail.domain'

### **Use smtp instead of sendmail/postfix.**
gitlab_rails['smtp_enable'] = true
gitlab_rails['smtp_address'] = "smtp.youremail.domain"
gitlab_rails['smtp_port'] = 25
gitlab_rails['smtp_user_name'] = "username"
gitlab_rails['smtp_password'] = "password"
gitlab_rails['smtp_domain'] = "youremail.domain"
gitlab_rails['smtp_authentication'] = :login
gitlab_rails['smtp_enable_starttls_auto'] = true
gitlab_rails['smtp_tls'] = false
```

上面的 **Email** 配置按照邮箱账号配置。使用如下指令更新配置并重启:

```
sudo gitlab-ctl reconfigure
```

如果要对外提供服务,可在防火墙的流量策略中新增一条如下所示的规则:

```
192.168.1.20:80→10.10.10.20:80
```

使用浏览器访问 **GitLab**, 管理员的用户和初始密码如下所示:

```
Username: root
Password: 5iveL!fe
```

启动与停止服务的指令如下所示:

```
gitlab-ctl start|stop|status|restart
```

社区版的安装说明还可以参考如下链接的中文说明:

```
https://www.gitlab.com.cn/installation/#centos-7
```

14.3 Redis 安装

我们使用高性能的 Redis 数据库来提供缓存服务，这不但可以避免 MySQL 数据库的主从同步延迟所引起的数据不一致问题，而且还能提高数据的访问性能。

如下网址可以下载 Redis 的安装包：

```
http://download.redis.io/releases/
```

下面以安装“3.2.8”的版本为例进行说明。

按顺序执行如下指令即可完成安装：

```
wget http://download.redis.io/releases/redis-3.2.8.tar.gz
tar xzf redis-3.2.8.tar.gz
cd redis-3.2.8
make
make install
```

使用如下指令编辑 Redis 配置：

```
cp redis.conf /etc
vi /etc/redis.conf
```

将其中的“daemonize no”改为“daemonize yes”，保存并退出编辑。

使用如下指令启动：

```
/usr/local/bin/redis-server /etc/redis.conf
```

使用如下方法进行测试：

```
redis-cli
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> get foo
"bar"
127.0.0.1:6379>
127.0.0.1:6379> quit
```

设置开机启动，创建一个启动文件：

```
vi /etc/init.d/redis
```

输入如下内容：

```
#chkconfig: 2345 80 15
#description: Start and Stop redis
PATH=/usr/local/bin:/sbin:/usr/bin:/bin
REDISPORT=6379
```



```

SERVERIP=10.10.10.29
EXEC=/usr/local/bin/redis-server
REDIS_CLI=/usr/local/bin/redis-cli
PIDFILE=/var/run/redis_6379.pid
CONF="/etc/redis.conf"

case "$1" in
    start)
        if [ -f $PIDFILE ]
        then
            echo "$PIDFILE exists, process is already running or crashed"
        else
            echo "Starting Redis server..."
            $EXEC $CONF
        fi
        if [ "$?"="0" ]
        then
            echo "Redis is running..."
        fi
        ;;
    stop)
        if [ ! -f $PIDFILE ]
        then
            echo "$PIDFILE does not exist, process is not running"
        else
            PID=$(cat $PIDFILE)
            echo "Stopping ..."
            $REDIS_CLI -h $SERVERIP -p $REDISPORT -a 12345678 SHUTDOWN
            while [ -x ${PIDFILE} ]
            do
                echo "Waiting for Redis to shutdown ..."
                sleep 1
            done
            echo "Redis stopped"
        fi
        ;;
    restart|force-reload)
        ${0} stop
        ${0} start
        ;;
    *)
        echo "Usage: /etc/init.d/redis {start|stop|restart|force-reload}" >&2
        exit 1
esac

```

使用如下指令设为开机启动:

```
chmod +x /etc/init.d/redis
chkconfig --add redis
chkconfig redis on
```

14.4 RabbitMQ 安装

安装 RabbitMQ 时需要 Erlang 支持, 这些安装包可以通过下列网址下载:

```
http://www.erlang.org/downloads
http://www.rabbitmq.com/releases/rabbitmq-server/
```

我们将使用如下两个安装包:

```
erlang-19.0.4-1.el7.centos.x86_64.rpm
rabbitmq-server-3.6.6-1.el7.noarch.rpm
```

安装支持环境:

```
rpm -ivh erlang-19.0.4-1.el7.centos.x86_64.rpm
yum -y install socat
```

安装 RabbitMQ:

```
rpm -ivh rabbitmq-server-3.6.6-1.el7.noarch.rpm
```

设置开机启动:

```
systemctl enable rabbitmq-server
```

编辑配置:

```
vi /etc/rabbitmq/rabbitmq.config
```

输入如下内容:

```
[{rabbit, [{loopback_users, []}]}].
```

启动服务:

```
service rabbitmq-server restart
```

开启管理功能:

```
rabbitmq-plugins enable rabbitmq_management
```

重启服务:

```
service rabbitmq-server restart
```

增加管理员用户:


```
rabbitmqctl add_user admin 123456  
rabbitmqctl set_user_tags admin administrator
```

在浏览器中使用端口“15672”登录控制台，可以对 RabbitMQ 进行管理，如下所示：

```
http://10.10.10.39:15672
```

14.5 小结

本章使用开源的 FastDFS 搭建了一个高可用的分布式文件系统，并通过 Nginx 为文件的访问设置了负载均衡服务，从而为微服务应用提供了一个高性能的文件服务器。另外，还简要介绍了 GitLab、Redis 和 RabbitMQ 等一些基础服务设施的安装和配置，其中，Redis 和 RabbitMQ 还可以根据需求建立集群服务。

到目前为止，电商平台所有用到的一些管理系统和服务设施都已经安装完毕，下一章将介绍怎么为微服务的发布提供持续交付和自动部署的方法。

使用自动化构建工具 Jenkins 实现 CI/CD

我们在前面的章节中曾经说过，使用微服务架构设计之后，一个应用平台将会有几十个到几百个应用的规模，这么多应用的测试和部署将会是一个非常巨大的工程，使用人力来完成这一方面的工作已经不太现实，所以我们要借助自动化工具来实现各个微服务工程的 CI/CD 工作流程。

CI/CD 为持续集成（Continuous Integration）和持续部署（Continuous Deployment）的总称，是指通过自动化的构建、测试和部署，实现软件产品可循环使用的快速交付流程。

Jenkins 是一个基于 Java 开发的功能强大的自动化构建工具，并且有一个非常丰富的插件仓库的支持，可以很方便地扩展其自身的功能。Jenkins 的官方网址如下所示：

```
https://jenkins.io/
```

打开上面链接后，页面如图 15-1 所示。

通过单击页面上的“Plugins”选项，可以查看各种插件的介绍，如图 15-2 所示。

使用 Jenkins 可以很容易地结合使用 Maven、Docker、Selenium 和 JMeter 等工具，建立一个持续交付的自动化设施。



图 15-1 Jenkins 官网首页

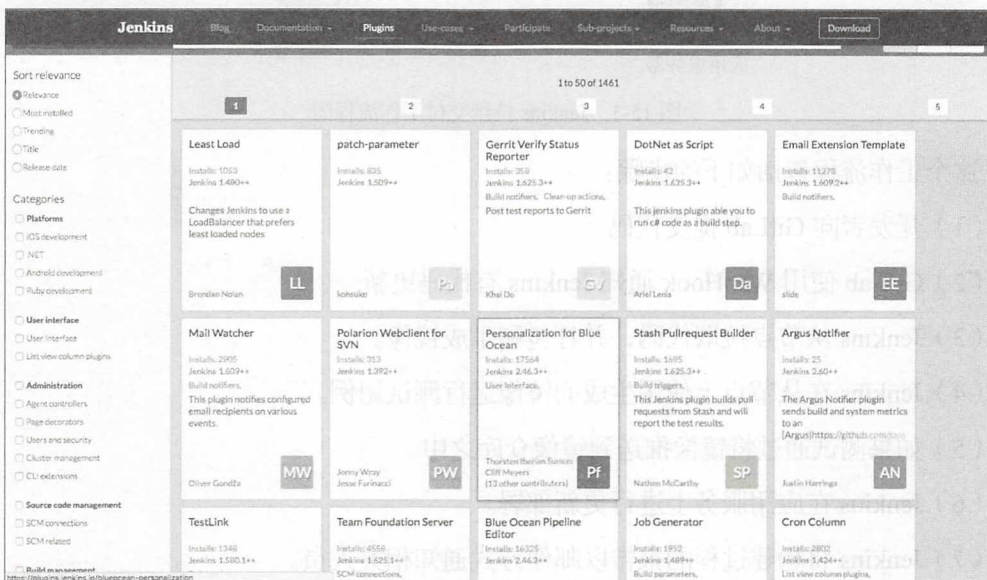


图 15-2 Jenkins 丰富的插件仓库

15.1 持续交付工作流程

使用 Jenkins 可以从代码提交开始,建立自动测试和自动部署的自动化流程,如图 15-3 所示。

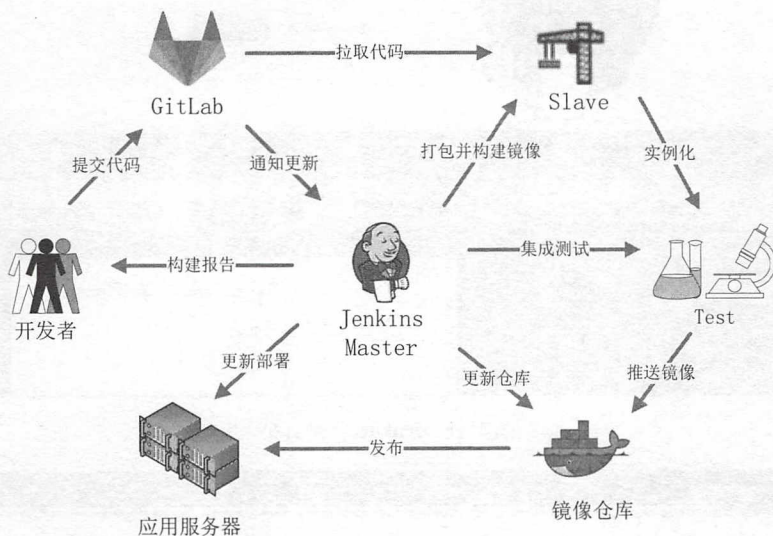


图 15-3 Jenkins 持续交付工作流程图

这个工作流程包括如下的步骤：

- (1) 开发者向 GitLab 提交代码。
- (2) GitLab 使用 WebHook 通知 Jenkins 有代码更新。
- (3) Jenkins 从节点拉取代码，并打包和生成镜像。
- (4) Jenkins 在从节点上使用生成的镜像运行测试用例。
- (5) 如果测试通过将镜像推送到镜像仓库之中。
- (6) Jenkins 在应用服务上进行更新部署。
- (7) Jenkins 将构建过程的报告以邮件方式通知相关人员。

从开发人员向代码库提交代码之后，整个流程都是自动进行的。如果中间哪个环节出现错误，将会中止流程的执行，并将结果通知相关人员。提交的代码不但要包括应用程序，还应该包括构建镜像的脚本、测试用例的脚本和部署的编排脚本等。

其中，各个步骤的操作可以使用插件或直接在命令行中使用各种工具来完成。

例如，拉取项目代码将会用到 **Git** 插件，项目的打包将使用 **Maven** 工具来实现，生成镜像和应用部署将直接通过命令行来使用 **Docker-compose** 工具，而集成测试也将通过命令行来执行由 **Selenium**、**JMeter** 等工具生成的脚本。

下面使用一个简单的实例说明 **Jenkins** 的使用方法。

15.2 Jenkins 安装

下面的安装过程以 **MacOS** 为例进行说明。

Jenkins 需要 **JVM** 的支持，请确保你的机器上已经安装了 **JDK 1.8** 或以上版本。为了完成后面的自动化演示，请确认你的机器中已经安装了 **Maven**、**Git** 客户端和 **Docker**。

打开如下网址下载安装包：

<https://jenkins.io/download/>

如图 15-4 所示，选择左边的 **LTS** 稳定版中的 **Mac OS X** 版本下载。

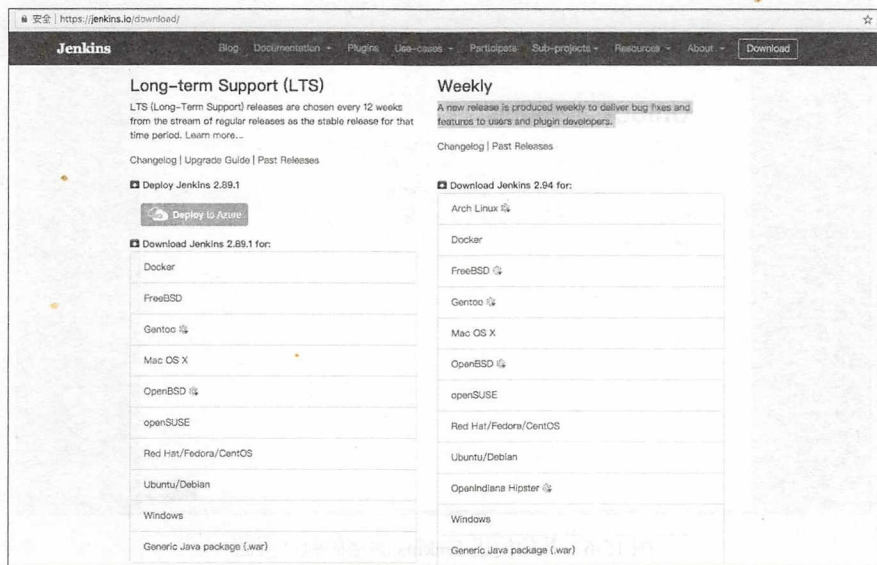


图 15-4 Jenkins 安装包下载页面

下载完成后，单击安装包“jenkins-2.89.1.pkg”开始安装，如图 15-5 所示。

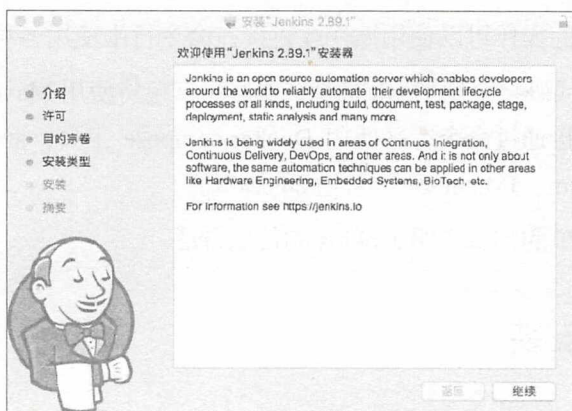


图 15-5 安装 Jenkins 的欢迎界面

安装过程比较简单，直接单击“继续”按钮，按提示选择系统推荐的插件。完成安装后，通过如下网址打开本地的 Jenkins：

`http://localhost:8080`

第一次打开后将会看到如图 15-6 所示的页面，进行 Jenkins 的密码验证。

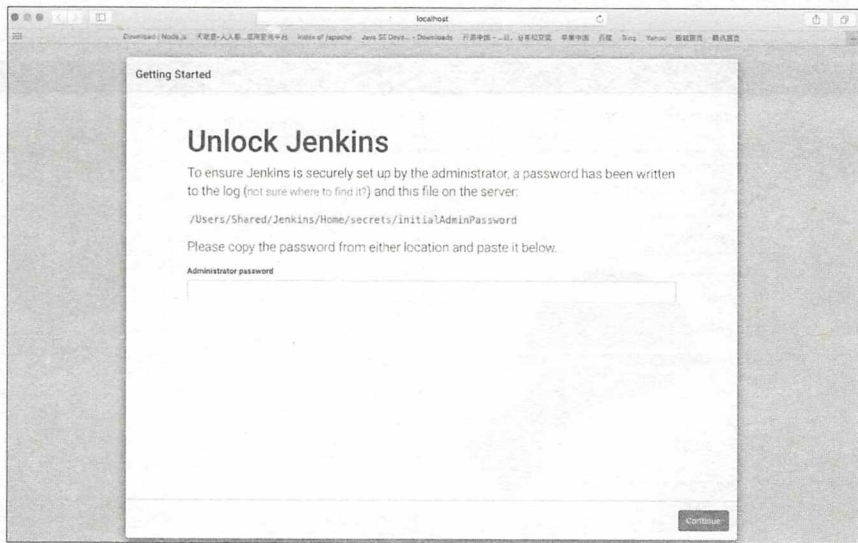


图 15-6 开始使用 Jenkins 的密码验证页面

按如图 15-6 所示的提示打开管理员密码文件，复制密码后，粘贴到密码输入框中，然后单击右下角的“Continue”按钮。如果密码验证成功，将会提示你创建一个操作员用户。创建一个操作员用户后，登录 Jenkins，即可开始使用，如图 15-7 所示。



图 15-7 新用户登录的欢迎界面

15.3 Jenkins 基本配置

由于要用到 Maven 编译和打包，可在欢迎界面的“系统管理”中打开“全局工具配置”，如图 15-8 所示。



图 15-8 Jenkins 系统管理操作界面

在“全局工具配置”中单击“Maven 安装”，设定一个名字，并设置 Maven 的安装路径，如图 15-9 所示。

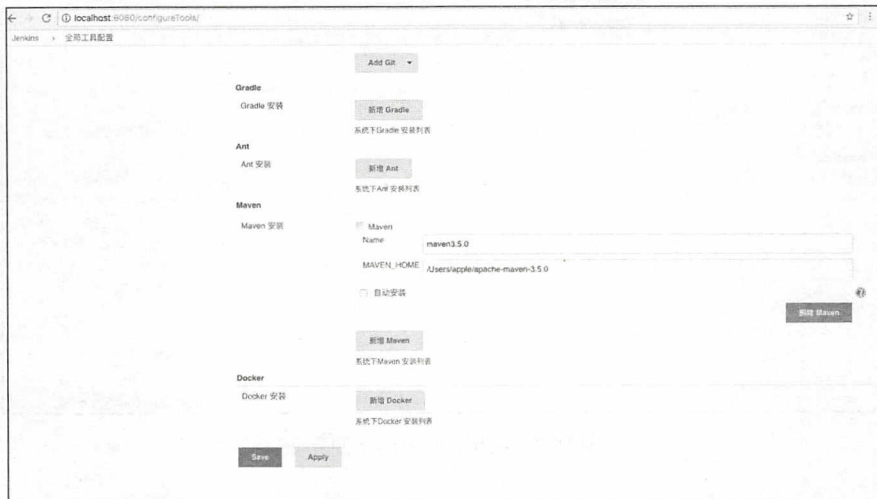


图 15-9 Maven 配置操作界面

在“系统管理”操作界面中单击“管理插件”，从“可选插件”中选择“Maven Invoker Plugin”插件，单击底端的“直接安装”按钮，如图 15-10 所示。

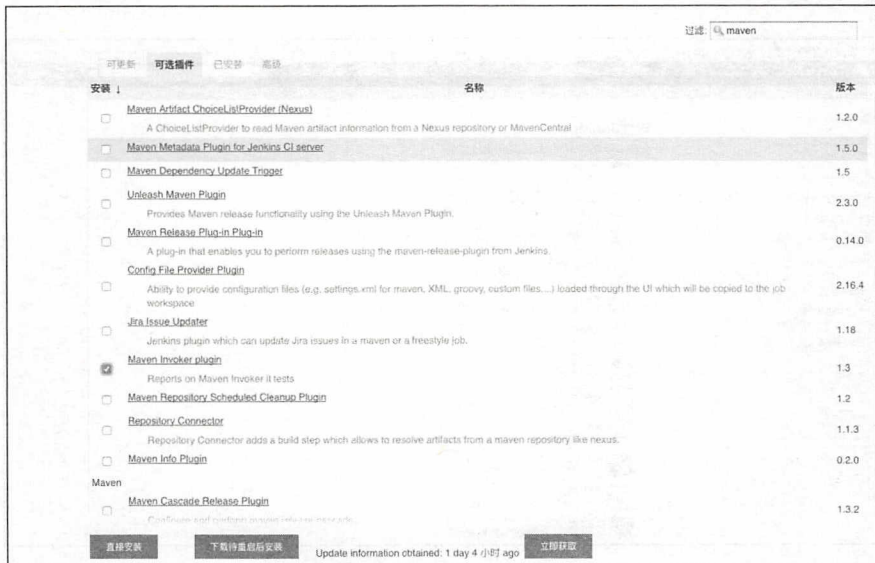


图 15-10 Jenkins 管理插件操作界面

注意：Maven 的 settings.xml 配置中的 repositorys 路径的设置，如果是在本机测试最好与 IDEA 的配置相同，这样打包时将不用再重新下载一次依赖包。

为了在 Jenkins 的命令行配置中能够正常使用 Docker 和 Docker-compose 工具的指令，需要对 Jenkins 的系统权限进行一些设置。因为 Jenkins 程序使用默认用户“jenkins”来开启服务，所以权限设置就是为这个用户进行授权。

通过如下操作步骤，为“jenkins”用户设置一个免密码配置，这样，在 Jenkins 的命令行配置中就可以使用超级管理员的指令“sudo”。

在 MacBook 的终端中执行如下指令，切换到超级管理用户即 root 中，并输入 root 的密码：

```
appledeMacBook-Air:/ apple$ su
Password:
```

在超级用户状态中编辑“sudoers”，并找到如下所示信息：

```
sh-3.2# vi /etc/sudoers
# root and users in group wheel can run anything on any machine as any user
root          ALL = (ALL) ALL
%admin        ALL = (ALL) ALL
```

在上面信息的后面，参照 root 的权限设置添加如下所示的配置并保存：

```
jenkins ALL=(ALL) NOPASSWD: ALL
%admin  ALL=(ALL) NOPASSWD: ALL
```

然后，使用“dscl”指令将 jenkins 用户加进 admin 用户组中，这个指令等同于一般 Linux 操作系统“usermod”的用法：

```
sh-3.2# dscl . -append /Groups/admin GroupMembership jenkins
```

这就完成 Jenkins 的权限设置。

15.4 Jenkins 自动部署实例

现在，为了演示 Jenkins 的使用，我们来创建一个自动部署的实例。

这个实例使用的是一个功能非常简单的项目，项目存放在码云的 Git 代码库中，如下所示：

```
https://gitee.com/chenshaojian/demo.git
```

项目中只有一个主程序，代码如下所示：

```
@SpringBootApplication
@RestController
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @RequestMapping(value = "/")
    public String index(){
        return "Hello World!";
    }
}
```

即应用启动后，打开首页将输出“Hello World!”信息。

下面说明这个项目的自动部署的实现过程。

15.4.1 创建任务

在 Jenkins 的首页中单击“新建”，打开一个创建任务的页面，如图 15-11 所示。



图 15-11 创建任务页面

在“输入一个任务名称”文本框中输入“demo”，并选择“构建一个自由风格的软件项目”，然后单击左下角“确定”按钮，创建了一个空任务，如图 15-12 所示。



图 15-12 创建 demo 任务

15.4.2 任务配置

在上面创建的 demo 任务中，单击“源码管理”选项，选择“Git”，并在代码库的地址中输入“demo”项目的存放地址，如图 15-13 所示。



图 15-13 源码管理配置

因为这是一个公开项目，所以不用设置访问项目的权限。如果是一个私有项目，必须在如图 15-13 所示的“Credentials”中配置对项目有存取权限的用户名和密码。

单击“构建触发器”选项，勾选“Poll SCM”，可以配置一个定时任务的日程表，如图 15-14 所示。

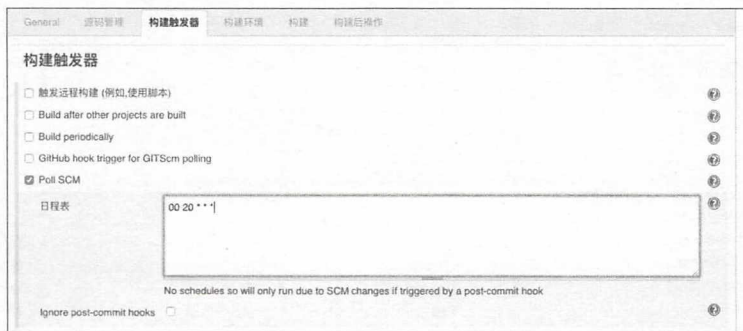


图 15-14 定时任务配置

图 15-14 中的日程表“00 20 * * *”表示在每天的 20:00 点整执行任务构建。

使用 Maven 配置项目的打包，单击“构建”选项，在“增加构建步骤”中选择“Invoke top-level Maven targets”，如图 15-15 所示。

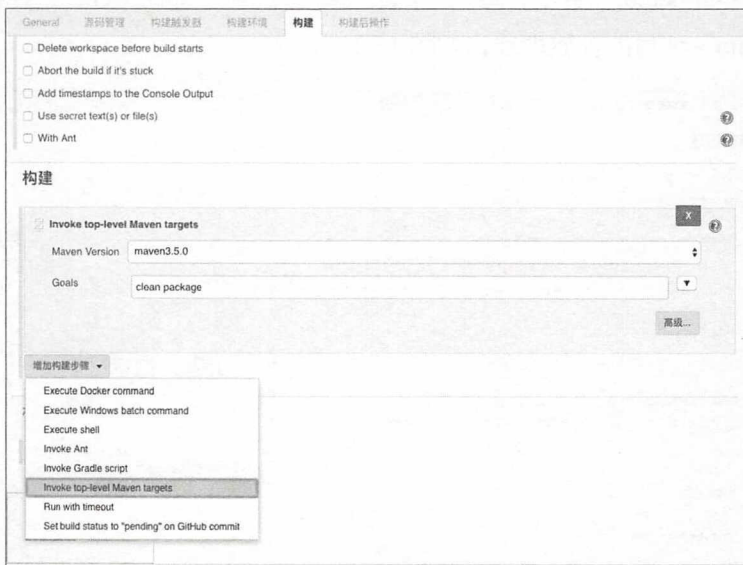


图 15-15 Maven 打包设置

其中，在“Maven Version”中选择在前面安装的 Maven。在“Goals”中输入如下所示的打包命令：

```
clean package
```

接下来配置创建镜像和部署的操作指令，这里要使用到 Dockerfile 和 docker-compose.yml，这两个文件已经包含在项目工程的 docker 目录中。



其中，Dockerfile 的内容如下所示：

```
FROM java:8
VOLUME /tmp
ADD demo-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 8080
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

docker-compose.yml 的部署脚本如下所示：

```
demo:
  build: .
  ports:
    - "8888:8080"
```

在“构建”选项中，单击“增加构建步骤”，从中选择“Execute shell”，然后在“Command”中输入如下所示的指令：

```
cd /Users/Shared/Jenkins/Home/workspace/demo/docker
cp -f ../target/demo-0.0.1-SNAPSHOT.jar .
sudo /usr/local/bin/docker-compose down --rmi all
sudo /usr/local/bin/docker-compose up -d
```

这些指令与我们在第 11 章中使用的手动部署的方法是一样的，即先停止正在运行的容器，并删除容器和镜像，然后重新进行部署。完成设置后如图 15-16 所示。

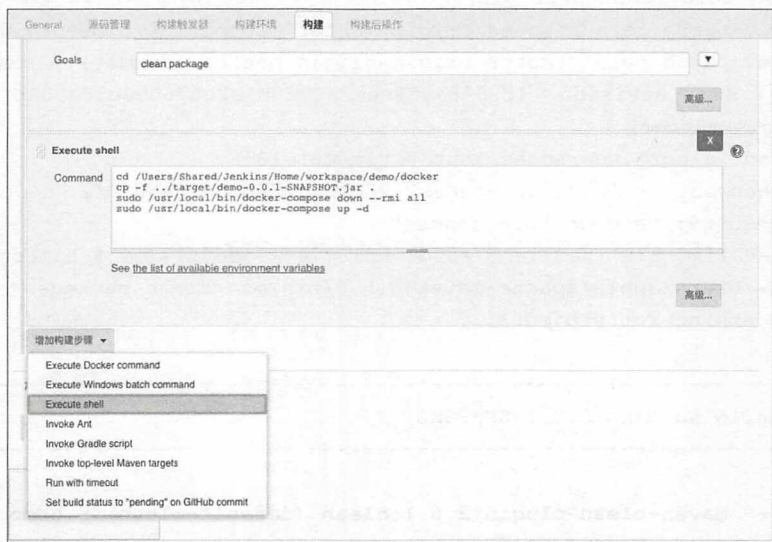


图 15-16 部署设置

15.4.3 执行任务

手动执行任务时，通过单击任务的名字返回任务的首页中，单击左侧菜单的“立即构建”即可开始执行，如图 15-17 所示。



图 15-17 在任务首页执行“立即构建”

任务的执行过程将在控制台中输出，如下所示：

```
Started by user mr.csj
Building in workspace /Users/Shared/Jenkins/Home/workspace/demo
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://gitee.com/chenshaojian/demo.git #
timeout=10
Fetching upstream changes from https://gitee.com/chenshaojian/demo.git
> git --version # timeout=10
> git fetch --tags --progress https://gitee.com/chenshaojian/demo.git
+refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision 1b0348a999cee3a1920b1b2c576b54e58a50abf2 (refs/
remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 1b0348a999cee3a1920b1b2c576b54e58a50abf2
Commit message: "add docker-compose"
> git rev-list 8791f0a371ab67a83d1005197744475de5f177df # timeout=10
[demo] $ /Users/apple/apache-maven-3.5.0/bin/mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building demo 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ demo ---
[INFO] Deleting /Users/Shared/Jenkins/Home/workspace/demo/target
[INFO]
```



```

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ demo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ demo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/Shared/Jenkins/Home/workspace/demo/
target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
@ demo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/Shared/Jenkins/Home/
workspace/demo/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @
demo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/Shared/Jenkins/Home/workspace/demo/
target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.20:test (default-test) @ demo ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ demo ---
[INFO] Building jar: /Users/Shared/Jenkins/Home/workspace/demo/target/
demo- 0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.8.RELEASE:repackage (default) @ demo ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 5.095 s
[INFO] Finished at: 2017-10-30T16:18:18+08:00
[INFO] Final Memory: 29M/182M
[INFO] -----
[demo] $ /bin/sh -xe /Users/Shared/Jenkins/tmp/jenkins4696633078670494346.sh
+ cd /Users/Shared/Jenkins/Home/workspace/demo/docker
+ cp -f ../target/demo-0.0.1-SNAPSHOT.jar .
+ sudo /usr/local/bin/docker-compose down --rmi all
Removing image docker_demo
Failed to remove image for service demo: 404 Client Error: Not Found ("No such
image: docker_demo:latest")

```



```

+ sudo /usr/local/bin/docker-compose up -d
Building demo
Step 1/6 : FROM java:8
----> d23bdf5b1b1b
Step 2/6 : VOLUME /tmp
----> Using cache
----> 64c36a425bbf
Step 3/6 : ADD demo-0.0.1-SNAPSHOT.jar app.jar
----> 1788813d23d2
Step 4/6 : RUN bash -c 'touch /app.jar'
----> Running in e4cfd4447b78
----> 2c44a754963b
Removing intermediate container e4cfd4447b78
Step 5/6 : EXPOSE 8080
----> Running in 95b96954618e
----> 8bc53f642637
Removing intermediate container 95b96954618e
Step 6/6 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar
/app.jar
----> Running in a192a418f4f1
----> 3a27629ceba9
Removing intermediate container a192a418f4f1
Successfully built 3a27629ceba9
Successfully tagged docker_demo:latest
Image for service demo was built because it did not already exist. To rebuild
this image you must use `docker-compose build` or `docker-compose up --build`.
Creating docker_demo_1 ...
Creating docker_demo_1
^-[1A^-[2K
Creating docker_demo_1 ... ^-[32mdone^-[0m
^-[1BFinished: SUCCESS

```

从控制台的输出日志中可以看到构建已经成功完成,我们可以通过如下网址打开应用运行的首页:

```
http://localhost:8888
```

打开上面链接后,可看到我们预期的结果,即输出“Hello World!”,如图 15-18 所示。

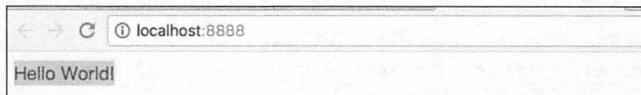


图 15-18 “demo”项目运行的首页



在上面执行任务的控制台输出日志中有一个如下所示的错误提示：

```
+ sudo /usr/local/bin/docker-compose down --rmi all
Removing image docker_demo
Failed to remove image for service demo: 404 Client Error: Not Found ("No such
image: docker_demo:latest")
```

出现这个错误提示是因为我们在第一次构建时并不存在可以移除的镜像，但这并不影响整个构建过程的执行。

现在来验证一下项目更新的自动化部署效果，我们可以将项目的主程序的输出结果“Hello World!”改为“Hello Jenkins”，然后提交代码。完成之后，再在 Jenkins 中单击“立即构建”，构建完成后，刷新访问应用的浏览器，即可以看到如图 15-19 所示的效果。

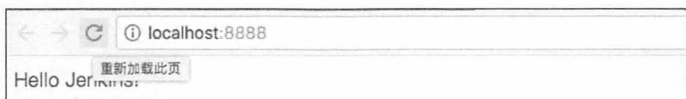


图 15-19 更新“demo”项目后的效果

再看看执行任务的控制台日志，现在，移除镜像的指令已经没有显示错误，而是输出了如下所示的结果，表示已经停止了运行的容器，并移除了原来的容器和镜像。

```
+ sudo /usr/local/bin/docker-compose down --rmi all
Stopping docker_demo_1 ...
^[[1A^[[2K
Stopping docker_demo_1 ... ^[[32mdone^[[0m
^[[1BRemoving docker_demo_1 ...
^[[1A^[[2K
Removing docker_demo_1 ... ^[[32mdone^[[0m
^[[1BRemoving image docker_demo
```

上面只是一个简单的自动部署的演示，我们使用了立即构建的方法来执行自动流程。在实际使用时，可以通过定时任务或结合使用 WebHook 的代码提交通知，来实现完全自动的部署流程。另外，还可以通过 Selenium、JMeter 等工具生成测试脚本，增加自动测试的功能。

15.5 小结

本章介绍了怎么使用自动化构建工具 Jenkins 来设计持续交付的工作流程，并以一个简单的实例演示了自动部署的实现过程，通过使用 Git 进行代码拉取、使用 Maven 进行程

序打包、使用 **Docker** 进行镜像的创建和应用的更新与部署，讲解了自动化流程中几个核心步骤的使用方法，从中可以看出 **Jenkins** 工具所具有的强大的可扩展性功能。通过本章的学习，相信读者能根据实际生产的情况建立一个完善的自动化基础设施，从而实现微服务发布中复杂的集成测试和持续部署的要求。

后 记

我们从微服务架构开始，使用一个电商平台实例，一起经历了架构设计、应用开发和部署的整个过程，不管读者从前有没有从事过微服务架构的设计和开发，有没有经历过高可用的服务器搭建的实践，有没有进行过自动化设施的建设，笔者都衷心希望，在这一过程的介绍和体验之中，本书能给你带来一些有益的帮助。因为本书的涵盖面较广，所以对某些方面的介绍和探讨可能还不够深入和详细，但笔者希望本书对读者来说可以是一个入门的指引，或者说能够激发读者在某些方面的兴趣，这样，再经过不断地探索和努力，就一定会有更大的成就。

微服务架构的设计理念已经深入人心，并且到处落地开花，硕果累累，而微服务的开发工具也正在日新月异的推陈出新之中，除了有众多 Java 开发者拥趸的非常活跃的 Spring Cloud 社区，还有其他很多优秀的团队一直在进行微服务的设计和开发的探索之中，例如，不久之前，华为也推出了开源的 Service Comb，Service Comb 是一个提供了一套包含代码框架生成、服务注册发现、负载均衡、服务可靠性（容错熔断、限流降级和调用链追踪）等功能的微服务开发框架。据说 Service Comb 还支持多语言开发，除了支持 Java，还支持 Go 等开发语言。详细的情况可以通过如下链接进行了解：

<http://servicecomb.io/cn/>

另外，一种支持微服务开发的新技术也正在悄然兴起，它的名字叫做 Service Mesh，中文翻译为服务网格。Service Mesh 是一个服务间通信的专用基础设施层，其功能在于提供安全、快速、可靠的服务间通信。详细情况可以使用如下所示的链接通过中文社区进行了解：

<http://www.servicemesh.cn/>

我们期待新技术的发展为微服务架构设计的实施提供更好的支持，同时也希望最终能真正做到，每个微服务的开发都可以轻易地使用任何一种开发语言来完成。



参考文献

- [1] 陈韶健. 深入实践 Spring Boot[M]. 北京: 机械工业出版社, 2016.
- [2] Sourabh Sharma. Java 微服务[M]. 卢涛. 译, 北京: 电子工业出版社, 2017.
- [3] 翟永超. Spring Cloud 微服务实战[M]. 北京: 电子工业出版社, 2017.
- [4] Spring Cloud[EB/OL]. <http://projects.spring.io/spring-cloud>.
- [5] Docker Documentation[EB/OL]. <https://docs.docker.com>.
- [6] Jenkins[EB/OL]. <https://jenkins.io/doc>.
- [7] James Lewis, Martin Fowler. Microservices[EB/OL]. <https://martinfowler.com/articles/microservices.html>.
- [8] Chris Richardson. Introduction to Microservices[EB/OL]. <https://www.nginx.com/blog/introduction-to-microservices>.

投稿联络：安娜

微信&QQ：80303489

邮箱：anna@phei.com.cn



Spring Cloud与Docker 高并发微服务架构设计实施

本书从架构设计、应用开发和运维部署三个方面出发，对微服务架构设计的实施进行了阐述和深入实践，并结合生产实际讲解了 Spring Cloud、Docker 和 Jenkins 等工具的具体使用方法。

书中通过一个互联网电商平台实例实现了高并发的微服务架构设计，并通过详细的开发和实施过程，演示了构建一个安全可靠、稳定高效并可持续扩展的系统平台的方法。

本书适合互联网应用开发设计人员参考学习。



博文视点Broadview



@博文视点Broadview



策划编辑：安 娜
责任编辑：牛 勇
封面设计：吴海燕

上架建议：微服务

ISBN 978-7-121-34161-8



9 787121 341618 >

定价：79.00元